Programmer's manual for SOS prototype - Version 4

Groupe Sor

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél (1) 39 63 55 11

Rapports Techniques

N° 103

Programme 3

PROGRAMMER'S MANUAL FOR SOS PROTOTYPE - VERSION 4

Le groupe SOR

Décembre 1988

RT_103

Manuel de Programmation du Prototype SOS Version 4

Programmer's manual for SOS Prototype Version 4

Technical Note SOR-35

The SOR group

INRIA, B.P. 105, 78153 Le Chesnay Cédex, France
tel.: +33 (1) 39-63-53-25
e-mail: sos@corto.inria.fr

December 1988

Abstract

This is a programmer's manual for SOS Prototype Version 4. It contains the necessary information to program SOS objects, in their different rôles (proxies, proxy-providers, and servers); how to prepare, export, or import proxies; how to prepare code objects; and how to compile. This paper contains the information on how to interface to the dynamic linker and to system services. This information is exposed in a compact form, by example; the reader is expected to have previous knowledge of C++ and of the principles of the SOS design.

Résumé

Ceci est le manuel de programmation du Prototype SOS Version 4. Il contient les informations nécessaires pour programmer les objets SOS, dans leurs différents rôles (mandataires, fournisseurs de mandataires, et serveurs); comment préparer, exporter ou importer des mandataires; comment préparer les objets codes; et comment compiler. Ce document donne des informations sur l'interface de l'éditeur de liens dynamique et des services systèmes, par l'intermédiaire d'exemples. Le lecteur est supposé connaître le langage C++ et les principes du système SOS.

About this document

This paper was written principally by Marc Shapiro, leader of the SOS task and of the research group "Systèmes à Objets Répartis" of INRIA, from material prepared by Sabine Habert. It also contains contributions from other people, and reflects the work of the whole SOS task:

- Dima Abrossimov is responsible for the kernel.
- Philippe Gautron is responsible for the C++ compiler and dynamic linker; he contributed material on compiling and debugging.
- Yvon Gourhant wrote the Speak application and did much of the initial debugging. He contributed Chapter 2, material on debugging and dynamic linker port on Metaviseur.
- Sabine Habert is responsible for the Acquaintance Service and wrote the initial material for this paper.
- Jean-Pierre Le Narzul, of Bull-MTS, is responsible for the Name Service; he contributed Chapter 9 and related material.
- Mesaac Makpangou is responsible for the Communication Service; he contributed Chapter 11.
- Laurence Mosseri, is responsible for the Object Storage Service; she contributed Chapter 10 and related material.

Other people working on SOS include Céline Valot and Vassilis Prevelakis.

Many thanks to all of them. Thanks also to our SOMIW partners who accepted and supported our work.

Typographical conventions

The following typographical conventions are used in this document. *Important* words and *defining uses* of a word are printed in italics.

Variable or procedure names, C++ code, keywords, or contents of administrative files are presented in **typewriter font**. Commands to type to the Unix shell are also in typewriter font, preceded by the prompt "**%** ", as in:

```
% sosman sosObject
```

References to the SOS Reference Manual are presented in SMALL CAPITALS. For instance the sentence "See sosCC[1]" means: look up the documentation of primitive **sosCC** in chapter 1 of the SOS Reference manual. See SOSMAN[1].

Contents

Chapter 1

Introduction

1.1 Goal and scope of this document

This Programmer's Manual is intended as the reference to programming in SOS for C++ programmers, in SOS Prototype Version 4. It gives simplified examples to introduce the concept of SOS objects, proxies, servers, proxy providers, etc. It shows how to write programs using these concepts, how to compile them, and how to use them at run-time. It also explains the interfaces to such system services as the Acquaintance Service, the Object Storage Service, the Name Service, and the Communication Service. It shows how the dynamic linker is used to migrate code objects; the creation of code objects is also explained.

This document explains how to program an application in the SOS system, Prototype Version 4. SOS is an object-oriented system, so this document is essentially on how to declare, define, use, and debug SOS objects. An object can play many rôles: a passive object, a client, a provider, a proxy or a server; at any point in time it can play one or many of these rôles. This document explains each one separately, from the simplest, to the most complex. Other concepts and support data structures are explained as needed.

This document and the *SOS Reference Manual* — which is available in hard-copy form [Sha87a] and on-line, see SOSMAN[1][1] — complement each other. The latter, a detailed reference of the meaning of SOS concepts and their usage, answers the question "What does this do?". This document is intended to answer the question: "How do I do this?".

[1] The notation TOPIC[CHAPTER] means: topic TOPIC is explained in chapter CHAPTER of the SOS Reference Manual. TOPIC may be the name of a program, an interface procedure, a system class, or an important concept.

Currently, CHAPTER may be 1 for Unix support programs, 2 for kernel or generic SOS object classes, functions, and C++ extensions, 3 for the Acquaintance Service, 4 for the Object Storage Service, 5 for the Name Service, 6 for the Communication Service, and 7 for Sos application programs and utilities.

Full detail on interfaces, allowable types of arguments, possible return conditions, and limitations may be found in the Reference Manual. This document contains some information from different sections of the Reference Manual, as well as additional background information.[2]

1.2 Proxies, Providers, and Servers

The SOS system is organized around the Proxy Principle [Sha86b]. An application program which wants to make use of some resource, a *client* of the resource, needs a local object which it can invoke, which is a representative of the resource. This representative is called a *proxy*, and is normally imported from the resource at the time of need by requesting it from a *proxy provider*. The proxy is a real object, with local procedures. It may elect to reply to invocations directly. It may also request more information from a *server* for the resource, executing in some other context.

From the system's point of view, there is no difference between a proxy and a server. The system only knows of *groups* of objects which can communicate by *cross-invocation*, via their *trap references*.

A proxy is a client's private interface to a resource; it usually was imported from a proxy provider, and carries one or more trap references towards servers. It may itself be a target of cross-invocations (e.g. from its server).

Proxies provided to different clients may behave differently, but they should all present a compatible interface.

1.3 Background

Previous knowledge of the SOS system and of C++ is required before reading this document. Please refer to [Sha86b], [Sha86c], [Sha86a], [Sha87b] for general information on SOS, and to [Sha87a], and [Gau87] for more specific information about the prototype. A full documentation packet will be sent upon request.

The best source about C++ is [Str85]. The programming language used for SOS is C++ with two extensions. *Exceptions* are non-sequential exits from a procedure or a block in order to signal errors (see EXCEPTION[2] and [Abr87]). *Dynamic classes* allow to use external objects (e.g. imported or permanent); they are explained in detail in [Gau87][3] and in the rest of this paper. *Reference* means an instance of the SOS class **ref**. There is no relationship with the C++ reference concept.

[2] If you notice any contradiction between these sources, please tell us. The on-line version of the Reference Manual can be considered the more credible source.

[3] Please note that the *inner* concept discussed in that paper is not used in SOS Prototype Version 4.

Please note that the SOS kernel is based largely on the "task" library of C++ [Str87]. All the primitives of this library are available for synchronization between concurrent tasks in the same context.

1.4 Organization of the Programmer's Manual

This document is organized as follows. The first few chapters explain how to use existing resources:

- Chapter 2 says how to set up the SOS environment and run an existing application.
- Chapter 3 explains the concept of objects in SOS, and especially migratable objects. Associated concepts like object naming and references are also explained.
- Chapter 4 explains the programming of a client application. A client is an application which uses some external resource; to do so it must import a proxy for that resource. This chapter explains how importation is done and how proxies are used by their clients.

The above chapters are sufficient for an application programmer who only wants to use existing resources. However, a programmer who wishes to program a resource for general usage must be aware of the distribution of proxies, and must know how to program proxies, providers, servers, and how they interface with each other. This is explained in the subsequent chapters.

- Chapter 5 explains how cross-context communication (between a proxy and its server) occurs. It explains the group concept, the **invokeMessage** and **returnMessage** data structures, and the segment concept. It also explains how a server receives and serves cross-invocations.
- When an object is migrated across a context boundary, there must be code in the target context to execute the object's invocations. It is carried in a **code** object, which is explained in Chapter 6.
- The proxy itself is the most complex part of programming a distributed application. Chapter 7 explains the life-cycle of a proxy, from creation and preparation by the provider, to exportation and re-initalization in the client's context, communication with its servers, and destruction. It also contains hints for debugging a proxy.
- Providers are objects which provide proxies to clients requesting them. Providers must create or select the proxy, and prepare it in a well-defined way. This is explained in Chapter 8.

The final chapters are targeted to application programmers with special needs. Chapter 9 gives information about giving symbolic names to objects and using those names. Chapter 10 explains how to create persistent objects and how to use them. Chapter 11 is about the various communication facilities available in SOS Prototype Version 4.

A glossary of useful terms can be found in Appendix A.

Appendix B is about changes between different versions (V1 to V2, V2 to V3, V3 to V4).

Appendix C deals with compiling and debugging of SOS applications.

1.5 The example

In order to illustrate different aspects of SOS, we use throughout this document, a simple example. It is called ''`Write`'' and is modeled after the Unix `write` program. This application allows a client to write on some other terminal.

A terminal manager ''`wrPrvdr`'' delivers proxies for clients for writing on a specific terminal. The terminal manager is in two parts: the `wrProvider` which delivers proxies, and the `wrServer` which does the connection to the terminal. There is one provider for all terminals, and one server per terminal.

This example runs on a single machine, and shows how one can program the proxy, the server and the proxy provider in this simple case, and explain other aspects of SOS. The entire example is presented in Appendix D.

The "speak" example is used in chapter 11 to illustrate remote communication and multicast within SOS.

Chapter 2

Running SOS and SOS applications

Prototype Version 4 runs on top of Unix. To start SOS, first of all, you must set up the SOS environment. This entails: setting up Unix environment variables if needed, creating a **predefContexts** file and an **sosDsk** sub-directory in the working directory, adding two entries in **/etc/services**, and then running the program **sos**. Then, you can run your applications.

In the following examples, we suppose that the SOS system is installed in the Unix directory **/sos**[1], and that you wish to run SOS within the working directory **/sosRun**.

2.1 Unix environment

First, the Unix environment variable **DLPATH** must be set with a Unix directory list, to search for exported code files used by the dynamic linker. Furthermore, your **path** or **PATH** environment variable should contain the directory for SOS binaries. For example, from **csh**:

```
% setenv DLPATH ".:/sos/export"
% set path = ( /sos/bin $path )
```

From the Bourne shell **sh** the syntax is:

```
$ DLPATH=.:/sos/export
$ PATH=/sos/bin:$PATH
$ export DLPATH PATH
```

[1] In our installation, this is a link to /users/SOS-CURRENT-VERSION, which is itself a link to /users/sos/v4, which is shared by NFS. The name of the installation directory (here, **/sos**) is a compile-time option.

2.2 The predefContexts file

The **predefContexts** file lists system services which must be run automatically when SOS is started. It must exist in the directory **/sos/etc**. The syntax of each entry is the name of a context, followed by a predefined integer which serves to compute its context's OID. If SOS binaries are installed in **/sos/bin**, the file **/sos/etc/predefContexts** looks like this:

```
/sos/bin/ASmain     1
/sos/bin/SSmain     2
/sos/bin/NSmain     3
/sos/bin/CSmain     4
```

Ordinary users should not change the **predefContexts** file. The absolute path **/sos/etc/predefContexts** may be overridden using the PREDEFCONTEXTS environment variable.

2.3 Storage Service environment

A Unix directory called **sosDsk** must be created in **/sosRun**. A subdirectory with the name of the machine[2] must also be created in **sosDsk**. For instance, to run SOS on a machine named "adele", you need to have the directories :

```
/sosRun/sosDsk
/sosRun/sosDsk/adele
```

Here is an example configuration in a system with 6 machines connected by NFS, where the same directory **/sosRun** is used as the working directory on all machines:

```
% cd /sosRun/sosDsk
% ls -ls
1 drwxrwxrwx  2 shapiro       512 Jan 14 10:12 adele
1 drwxrwxrwx  2 shapiro       512 Jan 14 10:12 averell
1 drwxrwxrwx  2 shapiro       512 Jan 14 10:12 blueberry
1 drwxrwxrwx  2 shapiro       512 Jan 14 10:12 carmen
1 drwxrwxrwx  2 shapiro       512 Jan 14 10:12 corto
1 drwxrwxrwx  2 shapiro       512 Jan 14 10:12 iznogoud
```

The absolute path **/sos/sosDsk** may be overriden with the SOSDSK environment variable. In case of problems, delete the files **/sosRun/sosDsk/*/*** and restart SOS.

[2] I.e. the result of the Unix **hostname** call.

2.4 Communication Service environment

In order to support distribution, one has to register two services in the Unix file **/etc/services**: **SOS-CS** and **SOS-TIMER**. The following definitions are used in our environment:

```
% grep SOS /etc/services
SOS-CS          2000/udp        csctxtPort      # SOS internal
SOS-TIMER       2001/udp        timerPort       # SOS internal
```

(If the port numbers 2000 and/or 2001 are not free in your environment, replace them by any other free numbers.)

Furthermore, you must register the list of potential SOS hosts, in the file **/sos/etc/sosHosts**. A host running SOS is capable of communicating only with those machines listed in its own **/sos/etc/sosHosts** file. Beware that the **sosHosts** files of all communicating hosts must be identical.

For debugging, you can restrict communication to a small set of machines by carefully editing **sosHosts** on each of the machines of the subset.

The pathname **/sos/etc/sosHosts** may be overridden with the SOSHOSTS environment variable.

2.5 Starting SOS

Once the above-described environment is set, start SOS kernel by running the **sos** program, e.g.:

```
% cd /sosRun
% sos &
```

Only one instance of the **sos** program may run at a time on a given workstation. **sos** may be run with arguments, for debugging (see Appendix C).

2.6 Starting applications

The services registered in the **predefContexts** file are started automatically by the **sos** program. Their argument vector **argv** is a copy of the arguments to **sos**.

Applications may be started, once **sos** is running, from a Unix program, e.g. the shell or a debugger. Just use its Unix pathname.

For instance, to run the "Write" program, you must first run **sos** with the standard environment; then run the "Write Provider", **wrPrvdr**; then run the **Write** program, once per client. So, once you start the SOS environment:

```
% cd /sosRun
% sos &
```

Then, start a single copy of the proxy-provider for Write:

```
% wrPrvdr &
```

Then every participating user types, on his own terminal:

```
% Write ttyName &
```

where **ttyName** is the name of the concerned terminal.

In order to start an application running on different workstations, one should first start the SOS environment on every participating workstation.

2.7 Creating a new context

A program can fork a new, independent, context by using the SOS kernel primitive **newContext**. One of its parameters is the Unix filename of the SOS application to start.

```
main() {
    ...
    newContext( ..., "Unix-filename", ... );
    ...
}
```

See NEWCONTEXT[2] for more explanation.

2.8 Terminating

A context may terminate itself by executing the **exit(int)** procedure; the parameter indicates success (value zero) or failure (non-zero, usually a Unix error code). This terminates all of its tasks immediately, and the context exits. Destructors for static objects are run, but not for any automatic or heap-allocated variables.

To terminate a context from the outside, just send the Unix interrupt signal. This can be usually done by typing the control-C or DEL character, or by sending the signal **SIGTERM** :

```
% kill process-id
```

where **process-id** is the process number of the context to kill. This has the same effect as executing **exit(-1)** from within the target context.

To terminate a whole SOS session, send the signal **SIGTERM** to the **sos** program on each participating machine. Type:

```
% kill sos-process-id
```

where **sos-process-id** is the process identifier of the **sos** program on that machine. That terminates the SOS kernel and all currently running contexts. Each of these terminates immediately, without executing *any* destructors, and with a non-zero status code.

Chapter 3

SOS objects

A programmer may create many object types, some of which are for the application's internal use, and others intended for communication with the system or with other application programs. This latter type must be known to the SOS run-time system, and more specifically to the SOS distributed Object Manager called the Acquaintance Service. They will be called SOS objects, or just simply objects when there is no ambiguity.

3.1 The `sosObject` class

An SOS object is simply an instance of some class derived from `sosObject`. When the class is instantiated, the constructor for `sosObject` will be called automatically, which will take care of all the declaring and interfacing to the SOS system (it automatically calls the appropriate procedures of the Acquaintance Service).

3.1.1 Declaring and creating an SOS object

To use an SOS object, the header file `context.h` must be included. Here is an example:

```
#include <sos/context.h>
class myClass: public sosObject {
    const char* s;
    int         i;
  public:
    void setString (const char* str) raises (nullArg) {
         if (!str) raise (nullArg);
         else      s = str;
    }
    const char* getString () { return s; }
```

```
    myClass (const char* str, int ii) {
                    s = str;
                    i = ii;
    }
};
. . .
/* this is an SOS object */
myClass* myObject = new myClass ("Hello, world.", 1024);
```

The last command creates an initialized instance `myObject` of class `myClass`, by calling its constructor defined above. This automatically calls the constructor (with no arguments) of class `sosObject`, which initializes the internal structures (Acquaintance Descriptor, OIDs). If necessary, any of the other constructors of class `sosObject` can be called.

3.1.2 Invoking an SOS object

Using an SOS object is the same as using any other C++ object. Creating an SOS object is simply calling one of its constructors: this will allocate space for the data, call the constructors for the base class (including the constructor for `sosObject`, which declares the new object to the Acquaintance Service). Invoking an object is calling one of its procedures defined in its public interface. For instance, returning to the example in section 3.1.1 above:

```
myClass *myObject = new myClass ("Hello, world.", 1024);
...
myObject -> setString ("Goodbye, cruel world!");
printf ("%s\n", myObject -> getString ());
```

As there are no public procedures to set or get the `i` field of `myClass`, it cannot be read or changed once the object is created.[1]

3.2 Naming objects

There are many ways of designating an object. Within a context, the address of the object or the index of its Acquaintance Descriptor can be used. Across contexts, a global designation or *Object IDentifier* (OID) must be used; this may be completed by a location *hint* to form a *reference*[2]. Furthermore, a *symbolic name*-to-reference mapping may be registered with the Name Service.

[1] C++ also allows access to public fields of the object. As explained below (see §7.6), it is bad practise (for debugging reasons) to have public fields in an SOS object. For the same reasons, never give out a pointer to a field of an object.

[2] "Reference" in this document means an instance of the SOS class `ref`. There is no relationship with the C++ reference concept.

3.2.1 Adresses and AD indexes

A normal C++ program refers to an object by its address in its context.

The Acquaintance Service refers to an acquaintance by its index in the Acquaintance Descriptor table. Acquaintance Service procedures are available to convert from one to the other, see GETADDRESS[3] and GETDESCRIPTOR[3].

```
class myClass: public sosObject {...};
...
myClass* myObject = new myClass(...);
// myDescriptor is set with the index of myObject
short myDescriptor = AS -> getDescriptor (myObject);
// addr is set with the address of object
sosObject * addr = AS -> getAddress (myDescriptor);
...
```

In the above example, `myObject`, `myDescriptor` and `addr` (see also chapter 10) all designate the same object. `AS` designates the local representative of the Acquaintance Service.

3.2.2 OID's and references

An OID is a 64-bit number which identifies an object in the SOS universe, see OID[2]. Each object has its unique (in this universe) OID, or "concrete OID" which characterizes its memory segment. It also carries a list of "group OIDs". All objects, across contexts, which carry a same OID are in the same group. With respect to this common OID, all objects in the group are considered equivalent (they are all incarnations of the same distributed object).

Searching an object by OID could be very expensive. The association of an OID and a location hint (to speed up searches) is called a reference; see REF[2].

An importation converts a reference into a local object (known by address or by Acquaintance Descriptor index). One can create a reference by GETREFERENCE[3]. Here are some examples:

```
ref aRef;                                  // some reference
OID anOID;
......
aRef.getOID(&anOID);                       // extract OID field
sosObject *myObject =                      // import
    new dynamic (&aRef) sosObject (...);
short myDesc =                             // extract descriptor
    AS -> getDescriptor (myObject);
ref localRef;
AS->getReference (myObject, &localRef); // create ref
OID localOID;localRef.getOID(&localOID);// extract its OIDs
```

In this example, **aRef** and **anOID** both designate the same global object, and **myObject, myDesc, localRef,** and **localOID** all designate its local representation.

For more explanation, see ACQUAINTANCESERVICE[3].

3.2.3 Symbolic names and the SOS Name Service

The symbolic names are the SOS user-level names (see NAMESERVICE[5]). The Name Service maintains the mapping between symbolic names and references. It allows, among others things, to register (**NS -> addName**) and look up (**NS -> lookup**) the symbolic names of objects.

NS -> addName gives a symbolic name to an object and registers the mapping between that symbolic name and the reference with the Name Service.

NS -> lookup asks for the reference of an object, given its symbolic name. It supposes that the symbolic name has been previously registered within the Name Service.

Other functionalities supported by the Name Service are presented in Chapter 9.

In order to use the Name Service, one must first import its proxy (see Chapter 4). This is best done at the beginning of the program. Example:

```
#include <sos/context.h>

dirs * NS;
main() {
   // import Name service proxy
   NS = new dynamic("/nameServer") dirs(nullIR);  // see below
   ...
   // create object and register it
   sosObject *myObject;
   NS -> addName ("/myDirectory/someName", *myObject);
   ...
   // get a registered reference
   ref myRef;
   NS -> lookup ("/otherDirectory/subDir/randomName", &myRef);
   }
```

3.3 Migratable objects

For an object to be suitable for migration, its class must be declared **dynamic**. This insures that invocations generated by the compiler will be indirect procedure calls via a *procedure table*, which is filled by the dynamic linker.[3]

[3]See [Gau87], and section 6.1 in this document.

```
/* file myClass.h */
    dynamic class
myClass: public sosObject {
    ...
    };
```

Using an instance of a dynamic class is exactly the same as for any other class: you simply invoke its procedures. The extra indirection via the procedure table is inserted by the compiler, and the user need not be aware of it.

Such a class can have locally-created instances, like below:

```
/* myObject1 is created locally */
myClass *myObject1 = new myClass ;
```

It can also have instances which are imported (migrated) from elsewhere. For instance:

```
/* myObject2 is imported from elsewhere. Syntax explained below. */
myClass *myObject2 = new dynamic (xyz) myClass (ir);
```

The special syntax **new dynamic** is explained further on (§4.1.1).

Importing an object makes the importer a *Client* of the resource represented by the proxy; the client rôle is detailed in Chapter 4.

Instances of a dynamic class (either created locally or imported) may be exported (or re-exported) to other clients. A proxy exporter is known as a *Provider*; this rôle is explained in Chapter 8.

Chapter 4

Client applications

An application program which wants to make use of some resource, a *client* of the resource, needs a local object which it can invoke, which is a representative of the resource. This representative is called a *proxy*, and is normally imported from the resource at the time of need. Later, in Chapter 5 we show how a resource's server communicates with a proxy; in Chapter 7, we show how the proxy itself is programmed; and in Chapter 8 we will show how a resource prepares and exports a proxy. For now, we will look at how a client acquires and uses a proxy.

4.1 Importing an object

4.1.1 Syntax

For the client, proxy acquisition appears like a special case of instantiation. The following construct is used:

```
importRequest ir= ...;
myClass * myObject;
...
myObject = new dynamic (resource) myClass (&ir,...);
```

where **resource** is either a symbolic name or a reference for the resource provider. The **resource** argument may be omitted, in which case it assumes the default symbolic name **"/services/myClass"**.

The first argument of the constructor **myClass (&ir...)** is a pointer to an instance of class **importRequest** (or of some derived class) [1]. The client should initialize the **importRequest** in the manner required by the resource.[2]

[1] Class **importRequest** is derived from the class **invokeMessage** and its size is limited to **MAXMESSAGE**.

[2] In future versions, this might be performed automatically by a stub compiler.

The `importRequest` is used to carry importation arguments; the pointer may be null if there are no arguments.

4.1.2 Re-initialization of an imported object

Importation is different from usual instantiation, in that the first argument to the constructor `myClass (&ir...)` *must* be, if not null, a pointer to an `importRequest` (or derived).[3]

This does not allocate a new instance. Instead, an initialized instance is imported from the resource, and re-initialized in the client's context by calling the importation constructor. This in turn calls the constructor for the base class and for embedded objects; special care is taken by the programmer of the proxy so that this does not lose the imported information (see §7.3).

4.1.3 Importing from a provider or from storage

The `resource` parameter of the `new dynamic (resource)` construct, a symbolic name or a reference, designates an object which provides proxies for this resource.

The `importRequest` is forwarded (by the Acquaintance Service) to the provider. The latter prepares a "proxy candidate" (see chapter 8) which is then transported into the client's context.

The `resource` parameter above may also designate an object stored with the Object Storage Service (Chapter 10). In this case, the object's image is copied from the SOS disk and activated in the client's memory.

In both cases, the mechanisms are the same, and the client sees no difference.

4.1.4 Pre-requisites

Requesting the importation of some object `X` has the side-effect of importing other objects which are designated as the "pre-requisites" for `X` (see §7.4). These are imported using the same mechanism as described above. The pre-requisite is initialized using its contructor with a single argument, an `importRequest*`. It is given a copy of the `importRequest` used for `X`.

4.2 Using an imported object

An imported object is used just like any other object. It is accessed by invoking its procedures. See Chapter 3.

[3] The compiler forbids calling a constructor whose first argument is a `importRequest*` in any other circumstance than in an importation.

4.3 Compiling a client

Any SOS program is a client of the SOS services and runs in an SOS context. It should include the standard header file **context.h**:

```
#include <sos/context.h>
...
```

The code for a client is compiled with the command sosCC[1]. For instance:

```
% sosCC -o client client.c
```

This compiles the source file **client.c** into an executable SOS application named **client**.

As we will see (Chapter 6), the code for the proxy is loaded and linked with the client at run-time if necessary. It is more efficient and easier to debug if the code is linked statically:

```
% sosCC -o client client.c proxy.o
```

The imported proxy is allowed to call procedures defined or linked with the importer. In particular, the proxy may call library functions (for instance **printf** defined in the standard library **libc.a**). To this effect, the library should be linked with the client application.

By default, **sosCC** always links the standard libraries, i.e **libc.a**, **libC.a** and **libD.a**. Other libraries may also be linked (see appendix C).

4.4 Debugging a client

The client's execution depends on the correct execution of the imported proxy. Therefore it is wise, during the debugging phase, to replace any importation request with a local instantiation of a stub object, with the same interface as the proxy.

Dynamic linking of the proxy's code makes debugging harder because the environment is less stable, and because the Unix debuggers don't know how to read the symbol table. It is therefore wise to link the code statically when possible.

The dynamic library **libD.a** provides run-time information about the current state of imported proxies. This library is automatically linked when source files are compiled with **sosCC**. A complete description of the debug facilities can be found in DL_INFO[8] and in "On the use of the dynamic linker in SOS V4", in directory **doc/dlink** of the current distribution.

More, with either dynamic or static linking, procedure **Zdebug()** is available, to list the dynamic classes currently linked (see ZDEBUG[8]).

For instance:

```
#include "dyninfo.h"

main(){
  Zdebug();
  ....
}
```

The default output from this procedure goes to the standard error output (file descriptor 2).

4.5 Example

Let us use the example of an SOS version of the UNIX `write` command. A client is the program of a user who wishes to write to another user, importing a proxy. The procedures of the proxy are then used to send the message on the terminal of another user (procedure `Write`):

```
#include <sos/context.h>
#include "wrProxy.h"

...

/* Start a Write client, to send messages to another terminal.
 * Import a write proxy, from the provider.
 * Then wait for user input, and give to write proxy.
 */

main (int argc, const char** argv)
{
    ...

    // create request message, and ask for importation
    wrImportRequest args (tty);
    wrProxy *myWrProxy =
        new dynamic (wrProviderName) wrProxy (&args);
    cerr << argv[0] << " is ready (" << rcsid << ")\n";
    // if importation fails, the program will abort
    // due to uncaught exception

    // read user input & call proxy
    while (...) { // read
        ...
        myWrProxy -> Write (buffer, strlen (buffer)+1);
    }

    ...
```

```
}
```

The class **wrImportRequest** is derived from **importRequest** and has a constructor to initialize the message.

As explained above, the effect of this is to send a copy of import request to the proxy-provider named by the constant **wrProviderName**, invoking its **giveProxy** procedure. The resulting object is installed as **myWrProxy**; it is initialized by calling the constructor which takes an **wrImportRequest** as its only argument. The above example is strictly equivalent to the following syntax, where the mapping of symbolic name to reference has been made explicit:

```
wrImportRequest args;
const char* wrServer = wrProviderName;
ref tmp;
NS -> lookup (wrServer, &tmp);
wrProxy* myWrProxy = new dynamic (&tmp) wrProxy (&args);
```

Chapter 5

Cross-context communication

We have seen that a client communicates with a resource via a proxy, an object in the client's context, which represents the remote resource. The proxy in turn usually communicates, across the context boundary, with one or many objects of the same group, which we call its *servers*. This chapter explains the mechanism of this communication, so that in the later we can explain how to program the server (Chapter 5.5) and the proxy itself (Chapter 7).

Support for cross-context communication is provided by the kernel primitive `crossInvoke`. Other forms are shared memory[1], shared persistent data, and dependencies[2]. This chapter explains in depth the cross-context communication provided by the kernel.

A proxy and its servers must be part of the same group. A proxy can perform a cross-context invocation to any member of the group, for which it has a "trap reference".

5.1 Groups

A group is a set of co-operating elementary objects across context boundaries. A group is characterized by an OID; all of the members of the group carry this OID in their Acquaintance Descriptor.

[1] In SOS Prototype V4, there is no support for shared memory. However we encourage the use of shared memory as an efficient communication medium between the proxy and the server. Since the prototype is built on top of Unix, you can use the System V shared memory operations (see *shmop(2)*) in the Unix manual.

[2] Not implemented yet.

5.2 Trap References

A trap reference is a reference (see REF[2]); it is also the capability for an object to cross-invoke the designated object. A trap reference is set by the provider before the object migrates, and is updated by the migration process (see SETTRAPREF[2]).

An object carries within its Acquaintance Descriptor, a list of **trapRefs**. When invoking **crossInvoke**, one has to specify which **trapRef** is the target of the invocation (by default, the kernel assumes the first one).

5.3 The kernel primitive crossInvoke

The semantics of **crossInvoke** (see CROSSINVOKE[2]) is like a Remote Procedure Call: the caller waits until the callee returns. Two "stub" pieces of code are necessary to interface with **crossInvoke**: the caller's (or proxy's) piece must marshall the invocation arguments within an invocation message, and unmarshall the reply message; the callee's (server's) piece do the converse work. In order to exchange arguments and reply, the caller, and the callee use two base classes: **invokeMessage** and **returnMessage**. We describe the content of these classes below.

The **crossInvoke** primitive takes three parameters:

- A pointer to an **invokeMessage** (or some derived class), containing appropriate values.

  The bare **invokeMessage** contains kernel-supplied information, including the **opaqueField** of the trap reference, and application-supplied information, the **opCode** field set by (the programmer of) the proxy, with an identification of the procedure the server should execute. This field is set by the caller's "stub" piece of code, and is interpreted by the callee's "stub" piece of code.

- a pointer to an array of **segmentDesc** pointers (described below).

- an optional index in the caller's list of **trapRefs**. By default, the kernel assumes zero.

The primitive **crossInvoke** returns a pointer to a **returnMessage** (or to an instance of some derived class). The bare **returnMessage** contains kernel-supplied information. A programmer is concerned with the **retCode**. This is the counterpart of the **opCode**. This code is set by the server's "stub" procedure, and is interpreted by the caller, once receiving the reply.

It is up to the "stub" code of the caller to copy the arguments and the **opCode** for the cross-invocation into the **invokeMessage**, and, upon return, to decode the reply, possibly raising an exception, depending on the return value **retCode**.

The size of **invokeMessage** and **returnMessage** is limited to the constant **MAXMESSAGE**.

The two "stub" pieces of code, could be generated automatically by a stub compiler, like in [Jon85]. We have planned such a tool.

5.4 Segments

If a large size of data is to be passed, one may use *segments*.

A segment is an array of bytes, supporting some operations protected by capabilities. During a **crossInvocation**, segments can be passed in both directions with capabilities to allow reading and/or writing them (see COPYTO[2]). The most frequently used capabilities are:

- **sgCopyFrom** in order to allow the destination to read the segment. For example, if a proxy wants to send an object to the server, it allocates a segment with **sgCopyFrom** capability, and then, assigns the concerned object to this segment (see ASSIGN[2]). Then, the server will read the segment once it receives the invocation.

```
segmentDesc* segs[2];  // declare two segments
/*
 * First, we allocate a segment with maxSize size.
 * sgCopyFrom capability allows to apply the  "copyTo"
 * operation to this segment in the destination context.
 * Second, we assign "object" to this segment.
 */
segs[0] =  new segmentDesc(0, sgCopyFrom);
segs[0] -> assign ( object, sizeof(object) );
segs[1] =  0;  // the last segment must be null

crossInvoke( ..., segs );
```

- **sgCopyTo** in order to give right to the destination to write the segment. If a caller wants to get, upon return, an object from the callee, it can allocate a segment with **sgCopyTo** capability. The callee will copy the concerned object within this segment during the **crossInvocation**.

Others capabilities and operations supported by segments are described in the reference manuel (see SEGMENTDESC[2]).

5.5 Servers: Programming the stub Procedure

A server is an object which defines the **stub** procedure, and towards which some proxy has a trap reference.

A stub decodes the invocation message. Depending on its **opCode**, it will perform some action. When this action is finished, **stub** stuffs the results into a return message, and terminates, causing the return message to be sent. The stub catches all exceptions which its action might raise, and transforms them into appropriate **retCodes** in the return message. Code for the **stub** procedure of the "Write"server is given in the next section.

5.6 Example

Let us return to the example of the SOS "Write". A client uses the Write procedure of the object **wrProxy** to send the message to its server. This message is sent as a segment with capability **sgCopyFrom**, thus allowing the server to read the contents of the segment.

```
enum {wrSend, wrQuit};

        int
wrProxy::Write (char* msg, int len)
                raises (error)
                raises (closed)
{
    ...

    // create and initialize invocation message
    wrInvokeMessage args (wrSend, len);

    // put the message in a segment
    segmentDesc *buf[2];
    segmentDesc sd(0, sgCopyFrom);
    sd.assign(msg, len);
    buf[0] = &sd;
    buf[1] = 0;

    // cross-invoke server, and get answer
    wrReturnMessage* res =
        (wrReturnMessage*) crossInvoke (&args, buf);

    // decode answer
    if(res->retCode != wrOk)
        raise (error);
    return res->result;
}
```

The **wrInvokeMessage** is a derived class of **invokeMessage**. **wrSend** is the **opCode** for this request. The **wrReturnMessage** is derived from **returnMessage**.

On the server side, the following code is executed:

```
    void
wrServer::stub
    (invokeMessage* request,
     returnMessage* reply,
     segmentDesc** segs)
{
    // Casting the reply and request to their derivated
    // classes.
    wrInvokeMessage * req = (wrInvokeMessage*) request;
    wrReturnMessage * res = (wrReturnMessage*) reply;

    switch (request->opCode) {

    case wrSend: {              /* Send message to terminal */
        ...

        // map received segment to a character string
        const int size = req->size;
        char *msg = new char [ size ];
        segmentDesc segdesc (0, sgCopyTo);
        segdesc.assign (msg, size);
        segs[0] -> copyTo (&segdesc);

        // output  immediately
        ...

        // clean up & return
        delete [size] msg;
        res->result = tty->bad();
        if (res->result)
            res->retCode = wrFailed;
        else
            res->retCode = wrOk;
        break;
    }

    case wrQuit:                /* close connection */
        ...
        res -> result = 1;
        res -> retCode = wrOk;
        break;

    default:                    /* unknown operation */
        reply -> retCode = wrBadCodeOp;
    }
}
```

Chapter 6

Code objects

6.1 Code objects

We will now explain the concept of code attached to a migratable object.

Any object is composed of (i) internal data, and (ii) the code (the object's procedures) which can interpret or modify this data. Programs which use an object are allowed to do so only via its public procedures.[1] Only the code has knowledge of the layout of the data. Therefore, when migrating an object, it is necessary to migrate its code also; for it to be useable, it must be dynamically linked with the extant code in the target context.

6.2 An acquaintance and its code object

The Acquaintance Descriptor of an SOS object `myWrProxy`, of class `wrProxy`, contains a reference to an SOS object which we call here `code-for-wrProxy`, instance of class `code` (see CODE[2]). When an instance of `myWrProxy` is migrated to some other context, `code-for-wrProxy` is migrated too, and initialized in the destination context. The initialization procedure for class `code` runs the dynamic linker, and initializes the procedure table.

It would be wasteful not to share code objects when possible, i.e. between instances of the same actual class. Therefore, the system checks that `code-for-wrProxy` doesn't already exist in the destination context before migrating it.[2] The migration procedure for class `code` (see GIVEPROXY[2]) actually copies `code-for-wrProxy` into the destination.

When preparing an object for migration, you must call `setCodeRef` (see SETCODEREF[2]) to set the code reference in the Acquaintance Descriptor. We

[1] C++ also allows access to public fields of the object. As explained below (see §7.6), it is bad practise to have public fields in a migratable object.

[2] See *Prerequisites* in Appendix A.

return to this later (see §6.6).

6.3 Declaring and using a code object

Consider the non-dynamic instance **myWrProxy1** in the example below. It will be attached to whatever code object already exists for its class **wrProxy** at instanciation time. If none is loaded yet, it will be read in from a default location; this default may be changed by adding a name[3] after the keyword **dynamic** in the class declaration. For instance, the following declaration says that the SOS symbolic name for the code for class **wrProxy** (if none is present yet) is **"/myExport/wrProxy.code"** instead of the default **"/export/wrProxy.code"**:

```
// declare (in file "wrProxy.h")

dynamic ("/myExport/wrProxy.code") class
wrProxy: public sosObject {
        ...
};

// now instantiate

wrProxy* myWrProxy1 = new wrProxy;
```

Alternatively, one can call the dynamic linker explicitly before the first instantiation of that class. The following piece of code achieves the same result as the previous example (see SOSFINDCODE[2]):

```
// declare (in file "wrProxy.h")

dynamic class
wrProxy: public sosObject {
    ...
};

/*
 * invoke dynamic linker before any instantiation
 * or importation for this class
 */
 ...
 ref myCode;
 NS -> lookup ("/myExport/wrProxy.code", &myCode);
 sosFindCode ("wrProxy", &myCode);

 // now, instantiate
```

[3] Or a reference.

```
wrProxy* myWrProxy1 = new wrProxy;
...
```

The first argument to **sosFindCode** is the name of the class, the second the reference of the code object.

6.4 The relation of an object to its code

Two objects which have been declared of the same **dynamic class** do not necessarily execute the same code, because every object has its own procedure table. Its class of declaration, or "apparent class", controls the size of the table, and the name and type of the procedure corresponding to each entry; these will be identical for all objects of the same apparent class. However the contents of an entry is filled with a pointer into the object's code object, which may be different according to how it was instantiated and/or migrated.

In the example above, **myWrProxy1** was created locally using whatever code already exists for **wrProxy**. An other instance may be imported from elsewhere, possibly with its own code. Their interfaces must be compatible, but the actions may be different.

6.5 Creating a code object

The **code** object may be either created and permanently stored on disk, using the **makecode** utility (see §6.7), or created on-the-fly, in the provider. The latter case is described through the following example where the code of the class **wrProxy** is found in a Unix file named **wrProxy.o**:

```
code* wrCode;
wrCode = new code( "wrProxy", "wrProxy.o", 0 );
```

The list of filenames must be followed by a NULL pointer.[4]

6.6 Using a code object

A **code** object is attached to an object, before exporting it, using **setCodeRef**.

```
// candidate for exportation

wrProxy* candidate = new wrProxy(...);

ref wrCodeRef;
```

[4]Separate compilation is supported only via the **mergexport** utility to merge several source or binary files into a single importable binary file (see MERGEXPORT[1]).

```
// creates the reference for wrCode

  AS->getReference( wrCode, &wrCodeRef );
  candidate -> setCodeRef (wrCodeRef);
```

If not set by the proxy itself, the provider of **wrProxy** objects must set the code reference of the candidate to the reference of the stored code object:

```
ref   wrCodeRef;
wrProxy* candidate = new wrProxy(...);

NS -> lookup ("/export/wrProxy.code", &wrCodeRef);
candidate -> setCodeRef (wrCodeRef);
```

6.7 Storing a code object

The **makecode** utility is an interface to class **code** and to storage. Once a Unix **.o** file has been obtained, an SOS code object may be obtained, by running the **makecode** command in the SOS environment:

```
% makecode wrProxy wrProxy.o
```

The above command takes the Unix file **wrProxy.o** containing the compiled code for the dynamic class **wrProxy** and transforms it into a code object stored on the SOS disk under the name **/export/wrProxy.code** (see MAKECODE[7] for a full description).

Chapter 7

Programming the proxy

A proxy is an object which represents a remote resource for some client. It is located in the client's context. The client invokes the resource by local invocations of the proxy. The proxy is normally imported from the resource at the time of need, by requesting it from a provider.

The proxy is a real object, with local procedures. It may elect to reply to invocations directly. It may also request more information from a server for the resource, executing in some other context.

The proxy and its servers must be in the same group. The proxy normally communicates with a server by cross-invoking it; there is a **trapRef** from the proxy to each of its servers.[1]

A proxy has a complex life-cycle. It must first be created and initialized. Then it can be migrated, or else copied, to some client; there it is re-initialized. From there it may be re-exported (or else copied again) to another client. Finally it may be destroyed.

The system preserves **trapRefs** across migration. Also, it preserves its execution environment, based on the concept of pre-requisites, i. e. objects which must accompany the migrated object.[2] However, it is up to the programmer of the provider to set up the **trapRefs** initially; it is up to the programmer of the proxy to set up the pre-requisites.

The life-cycle of the proxy, and all the things a proxy programmer must know to program and debug a proxy, are the subject of the present chapter.

[1] They may also communicate by other means, e. g. shared memory or shared persistent data.

[2] In version 4, only code objects can be defined as pre-requisites.

7.1 The life-cycle of a proxy

The subject of this section is examining the different phases of the life of a proxy, from its initial creation, to migration, its functional existence as a proxy in its client's context, to its final destruction. The programmer of the proxy must be aware of the existence of these phases, and their different requirements.

A proxy is a migratable object, so it must be an instance of a **dynamic** class.

1. *Creation :*

   A proxy starts its life by an instantiation as an ordinary object, in the context of a proxy provider. This is a local creation, using a constructor without an **importRequest*** as first argument.

   However it is done with the idea of exporting it and using it in a client context. The constructor must prepare the environment and the internal data it will need in its future life as a proxy. For instance, a proxy in our SOS **Write** example will contain a local view of the server it is attached to via the server OID. It will need an environment consisting of its code object (see §6.1).

2. *Exporting :*

   The provider may now select the object thus created for exporting (see GIVEPROXY[2]). This entails setting its group OID and **trapRefs**, and deciding between **giveCopy** and **giveSelf** (see Chapter 8).

   The system takes care of the actual exporting and installation of the data in the target context. The object's pre-requisites, if not already present in the target context, are first migrated into it and initialized (and recursively for their pre-requisites). This ensures, in our example, that the code object is in the client's context before the SOS **Write** proxy is installed therein.

   The object's **trapRefs** and dependency links are preserved across exportation.

3. *Re-initialization at client :*

   Once the data for an object is imported into the client's context, either by an explicit import request, or implicitly as a pre-requisite, the system calls the object's constructor. In other words, the data is initialized at least twice: once when created, and once at import time. This re-initialization normally does very little work, for instance it may fill pointers with locally legitimate values. However there is no restriction, it might completely re-write the data, or allocate another data segment (and set **this**), or request more importations (by executing **new dynamic**).

4. *Activity :*

   Now the proxy is installed in the client's context.

The client may invoke it, and the proxy may call local procedures of the client.

The proxy may communicate with other objects in the same group. For instance, it may share memory with another proxy or a server on the same machine. Or it may invoke a server in some other context; a server is an object towards which the proxy has a **trapRef**, and it is invoked by the system primitive **crossInvoke**. This proxy may itself be cross-invoked (if it is the target of a **trapRef** from some other object in its group).

5. *Re-exporting :*

   Now this client may itself be a provider, and decide to re-export the same object. Be warned that this re-exported object must not be the destination of a **trapRef** : this **trapRef** is not re-initialized when the target migrates.

6. *Destruction :*

   The proxy may now be destroyed by the client executing **delete**. This will run the proxy's destructor.

   However, the proxy might also be destroyed accidentally, for instance if the client's context is killed or its machine crashes. Therefore it is wrong to depend on the execution of the destructor.

   For instance, the destructor should not call back the server to clean up, for two reasons. First, as indicated above, you cannot count on this actually happening. Second, the destructor is also called as a side-effect of exporting with **giveSelf**.

Each of these phases will now be examined in detail.

7.2 The allocation of a proxy in the provider's context

Consider our example of the SOS **Write**. The provider exports proxies to allow clients to send some text to someone else's terminal. It instanciates a server that is in charge of sending these messages.

In order to export an object, it must be created initially.[3] Thus, the proxy's class must define at least one constructor, which the provider can use to create a local instance. This constructor must *not* have a **importRequest*** (or derived) as its first argument.

[3]It is possible to export directly from disk to the client, or to re-export an imported object. But in all cases an initial copy must be created in some provider's context; this is the operation we are interested here.

It is good practise for this constructor to also prepare the proxy for exporting, by setting its code reference (see SETCODEREF[2]), and setting the other prerequisites (see §7.4).

For instance, in the SOS **Write** example, let us create the **Write** proxy. Suppose we have the following declarations (e.g. in an include file **wrProxy.h**):

```
...

dynamic class wrProxy: public sosObject {
    friend class wrProvider;

    OID serverId;
    char termName [maxTermName];
    wrProxy (OID&, const char*) // local constructor (for provider only)
        raises (tooBig);
public:
    wrProxy (wrImportRequest*); // importation constructor
    ~wrProxy();                 // destructor
    ...

    int Write (char*, int);     // write msg on terminal
    void Quit();                // close connection
    virtual void print();       // check state
};
```

The private **wrProxy(OID&, const char*)** constructor can only be called by the provider. The OID field of the created instance is initialized with the server OID and **termName** with the name of the terminal. Assuming the code object has already been created using **makecode**, the reference of this code object is searched within the Name Service, and the proxy's acquaintance descriptor is updated.

```
/* The creation constructor, for creating a proxy-candidate.
 * This is reserved to the provider.
 */
wrProxy::wrProxy (OID& id, const char* tty)
                raises (tooBig)
                raises (noCode)
{
    serverId = id;
    if (strlen (tty) > maxTermName)
        raise (tooBig);
    strcpy (termName, tty);

    // set code reference too code object
```

```
    ref wrCodeRef;
    begin
        NS->lookup (wrProxyCodeName, &wrCodeRef);
    except
        when (notFound)
            raise (noCode);
    end
    this -> setCodeRef (wrCodeRef);
}
```

7.3 The importation into the client's context

Proxy importing is triggered by the client doing a "dynamic" instantiation:

```
...
main (int argc, const char* argv[])
{
   ...
   char* wrProviderName = ....;

   // create request message, and ask for importation
   wrImportRequest args (tty);
   wrProxy *myWrProxy =
      new dynamic (wrProviderName) wrProxy (&args);

   // if importation fails, the program will abort
   // due to uncaught exception

   ...
}
```

The "dynamic" instanciation is characterized by the keyword **dynamic**, and by the fact that the first (in this case, the only) argument to the constructor is a pointer to an **importRequest** (in this case, derived from **importRequest**). This will request an importation from the named provider.

Beware that the proxy, which was initialized once before exportation, is being re-initialized again! Care must be taken so that the constructor does not overwrite useful values. For instance, the following is wrong:

```
wrProxy :: wrProxy (wrImportRequest* ir){} // WRONG
```

This calls the constructor, with no arguments, of the embedded object **serverId**, which will reset the data to some null value.

There are two solutions. The recommended solution is to have a constructor with a first **importRequest*** argument for all embedded objects:

```
/* importation constructor for the wrProxy:
* there is nothing to do.
* Remember to call the importation constructor of OID
* so that the field serverId doesn't get over-written.
*/
wrProxy::wrProxy (wrImportRequest* ir):
        serverId(ir) {};
```

This ensures the programmers of class **wrProxy** that the constructor should not overwrite the data.[4] This is safe, because a constructor with a first **importRequest*** argument cannot be called outside of an importation. All the standard SOS date structures (for instance **ref** and **OID**) are provided with this kind of constructor.

This is not always possible, for instance if a class is part of a library of standard classes which you can't or don't want to change. Then it is possible to use the "copy" constructor **X(X&)** [Str85, page 180]. This is not quite the same as "doing nothing" but usually works.

7.4 Pre-requisites

Migrating some object **X** has the side-effect of requesting the import of other SOS objects, the designated "pre-requisites" for **X**. These are migrated using the general migration mechanism. The only difference is that, whereas the initialization of **X** is done under the control of the C++ compiler, the pre-requisites are imported by the system itself. Therefore the choice of an initialization procedure cannot depend on the number or type of arguments. A pre-requisite must have a constructor which has a single argument, an **importRequest***; the pre-requisite is initialized by calling this constructor with a copy of the **importRequest** used in the importation request for **X**.

The pre-requisites are imported, and initialized, before the initialization of X takes place. The main use[5] of pre-requisites is for code objects (Chapter 6).

7.5 The proxy's activity

Different proxies may have the same interface, i.e. are allowed the same actions, but do not have the same rights. Some actions can be performed locally by the proxy, some others can be forwarded to the server using the procedure **crossInvoke**. In our SOS **Write** example, the user message is forwarded to its destination by calling the **Write** procedure of the proxy. This message is packed in a segment which is passed as argument of the **crossInvoke** procedure towards the server.

[4] Except to update context-dependent data. See also **permPtr** in Chapter 10.

[5] And in the Prototype Version 4, the only one.

```
/* The client asks to write a message on the terminal.
 * This cross-invokes the server.
 */
        int
wrProxy::Write (char* msg, int len)
                raises (error)
                raises (closed)
{
    // if I already did a Quit, don't invoke server
    if (serverId == 0)
        raise (closed);

    // create and initialize invocation message
    wrInvokeMessage args (wrSend, len);

    // put the message in a segment
    ...

    // cross-invoke server, and get answer
    wrReturnMessage* res =
        (wrReturnMessage*) crossInvoke (&args, buf);
    ...

    // decode answer
    if(res->retCode != wrOk)
        raise (error);
    return res->result;
}
```

The invocation and return message are respectively derived from type **invokeMessage** and **returnMessage** (see §5.6) :

```
// operation codes between wrProxy and its server
// (field opCode)

enum {wrSend, wrQuit};

// return codes (field retCode)

enum {wrOk=0, wrBadCodeOp=-1, wrFailed};

// parameters passed with cross-invocations

struct wrInvokeMessage: public invokeMessage{
    int     size;
    wrInvokeMessage (int op, int sz) { opCode = op; size = sz;};
```

```
    wrInvokeMessage (int op) {opCode=op; size=0;};
};

struct wrReturnMessage: public returnMessage {
    int result;
};
```

7.6 Debugging a proxy

It is hard to debug migratable objects. The following tips are provided to make life easier on yourself.

- *No public fields.* A migratable class should not have any fields that clients could read or change directly.

  Instead, member functions should be available to read or set the fields.[6]

- *When possible, link statically.* The code for a dynamic class can be linked either dynamically or statically. In the latter case, no functionality is lost, because the run-time system checks if the code it needs is there; if what was statically loaded is not the right code, the system reverts to dynamic loading.

  Performance is bound to be better in the static case, unless you loaded the wrong code.

  Debugging is easier in the static case, a static environment is easier to master. Furthermore, the existing Unix debuggers don't know how to access the symbol table of dynamically-linked code.

At any time, debug facilities[7] and the procedure **Zdebug** (see ZDEBUG[8]) can help you to check the already linked dynamic classes.

A same binary file may be either statically or dynamically loaded. The compiler's debugging option **-g** is useful only in the first case (standard debuggers don't support dynamic linking):

```
% sosCC -g -c wrProxy.c
```

This option is useful for debugging statically-linked code, but for dynamically-linked code only results in increasing the loading overhead enormously.

The C++ translator generation can also provide help for debug. The procedure tables of **dynamic** classes are printed when the option **-F** is used:

```
% sosCC -F wrProxy.c > wrProxy..c
```

[6] Inline procedures pose no problem, because the compiler ignores the inline attribute for dynamic classes.

[7] Described in DL_INFO[8] and in "On the use of the dynamic linker in SOS V4", directory **doc/dlink** of the current SOS distribution.

The output file **wrProxy..c** contains useful comments, with the index of the dynamic tables. For instance, for the dynamic class **wrProxy**:

```
/* wrProxy dynamic tbl:
  -2 (- D) :  _wrProxy__ctorFPCwrImportRequest___
  -1 (- D) :  _wrProxy__dtor
   0 (V -) :  _sosObject_giveProxy
   1 (V -) :  _sosObject_stub
   2 (V D) :  _wrProxy_print
   3 (- D) :  _wrProxy__ctorFRCOID__PC__
   4 (- V) :  _wrProxy__ctorFPCwrImportRequest___
   5 (- V) :  _wrProxy__dtor
   6 (- V) :  _wrProxy_getTermName
   7 (- V) :  _wrProxy_Write
   8 (- V) :  _wrProxy_Quit
   9 (V D) :  _wrProxy_print
*/
```

An extra indirection is generated for all method calls of a dynamic class. The first column is the index of the procedure in the procedure table, and must be the same as the index of this extra indirection in the caller code. Two characters appears between parenthesises. The first is V if the method is virtual, the second D if the method is dynamic. The rest of the line is the C name of the procedure. Virtual methods appear two times, for needs of pointers to method. Note that non-dynamic (but then virtual) methods can appear in this table when the dynamic class inherites from a non-dynamic class with virtual methods.

The negative indexes are for internal needs of the system.[8] They may be undefined.

You can also use the tracing facilities of the kernel (see Appendix C).

7.7 Optimization

When your code is perfectly running, a good improvement to binary files size is to recompile them with option **+z** (delete generation of dynamic method names) and without option **-g** :

```
% sosCC -c +z proxy.c
```

[8] This is where are stored the constructor and the destructor, for use in migrations triggered by the system (see §7.4).

Chapter 8

Proxy providers

To use a service, a client application has to import a proxy for that service : the client fills an **importRequest** message, and then calls the proxy's contructor, using the **new dynamic** construct. At this time, the **importRequest** message is automatically transferred to the provider's context, and the **giveProxy** procedure of the requested object is activated.

A proxy provider is an SOS Object which redefines the **giveProxy** procedure. This procedure can be split into two steps :

1. it decodes the **importRequest** message; depending on its contents, and on the identity of the requestor, it selects the appropriate object to be exported (see §8.1).

2. it sets up the group OID(s) and the trap reference(s) of the object and chooses the mode of exportation (copy or move), thus preparing it for the migration (see §8.2).

When **giveProxy** returns, the Acquaintance Service takes care of the actual migration and installs the proxy in the requestor's context.

8.1 Selecting a proxy candidate

An object which is to be exported as a proxy (which we will call a "candidate" in this chapter) must first exist, either in the provider's context, or in the SOS secondary storage (managed by the SOS Storage Service). In the former case, it may be an object that the provider has already imported or, it is a locally-created object.

8.1.1 Creating the proxy candidate

In order to create a candidate, the provider has to instantiate it by **new**; it is impossible to export an object allocated on the stack (a C++ automatic variable), in **static** storage, or embedded in another object; unpredictable results would occur. For instance, in the "Write" example :

```
void
wrProvider::giveProxy (const importRequest* ir, proxyDesc* result)
  raises(refused)
{
    wrProxy* candidate;
    OID serverOid = ...;
    const char * ttyName = ...;
    ...
    // instantiation
    candidate = new wrProxy (serverOid, ttyName);
    ...
}
```

If the code object for **wrProxy** is not yet loaded, the compiler searches it under a default SOS pathname. A non-default pathname can also be specified (see §6.3).

8.1.2 Retrieving a candidate from storage

When the candidate exists on the SOS storage (suppose, for example, we need to access a "mailbox" type), there are two cases to be considered : the provider doesn't need to do some special initialization or it does (for example, setting up its trap reference).

In the first case, the provider has just to fill the proxy descriptor with the reference of the stored candidate :

```
   void
mBoxProvider::giveProxy (const importRequest* ir, proxyDesc* result)
   raises(refused)
{
    ref myBoxRef;
    char const* myBoxName = ... ;

    ...
    NS->lookup (myBoxName, &myBoxRef);
    result->proxy = myBoxRef;
}
```

In this case, the Acquaintance Service takes care to forward the migration request in the storage context, and the steps described in the following section are skipped.

When the candidate needs some initialization, the provider must first import it from storage, then initialize it :

```
    void
mBoxProvider::giveProxy ( ... )
    raises(refused)
{
    ref myBoxRef=...;

    ...
    // importation + instanciation
    mailBox* myBox = new dynamic (&myBoxRef) mailBox (nullIR);
    myBox -> setTrapRef(Server);
    ...
}
```

8.2 Preparing the proxy candidate for migration

8.2.1 Creating group and setting trap reference

If a proxy has to communicate with a server, it must belong to the same group and have a trap reference to that server (trap references are only allowed within the same group). All members of the same group have a common OID. Providers create groups and update them with new members. A provider calls **giveMyOID** to create or extend a group, and it calls **setTrapRef** to make a proxy point to its server (and/or possibly vice-versa).

```
void
mBoxProvider::giveProxy ( ... )
  raises(refused)
{
    wrProxy* candidate;
    wrServer* server;

    ...
    giveMyOID (server, 1);
    // server acquires provider's group OID, so does candidate
    ...
    giveMyOID (candidate, 1);
    candidate->setTrapRef (server);
    // candidate's trap reference points to server
    server->setTrapRef (candidate);
    // and vice-versa
}
```

8.2.2 Setting code reference

Before exporting, the code reference of the candidate must be set. This is normally done by the candidate itself.

8.2.3 Choosing the mode of exportation

If the selected candidate is an instance of the provider's context, the provider has to decide between either exportation of a copy of the candidate or migration of the candidate itself. For each of the cases above, a separate function is provided, which fills a descriptor for the object (ready to be exported by the Acquaintance Service).

`giveCopy` causes a bitwise copy of the proxy to be exported and re-initialized in target context. The original remains in the provider's context (and may be re-exported) :

```
candidate -> giveCopy();
```

`giveSelf` causes the proxy to effectively migrate (copied to target context and re-initialized; the original is then destroyed in the provider's context) :

```
candidate -> giveSelf();
```

8.3 Example

We will show the example of a provider for the "Write" application. It will create and extend an appropriate group (by allowing the server and the proxy candidate to inherit its OID). It will finally set up a trap reference before exporting the candidate as the requestor's proxy.

```
/* Create a provider object.
*/
wrProvider::wrProvider(unsigned long seed)
{
    // allocation of a group OID for the Write service
    OID oid;
    oid.groupAllocate (seed);
    AS -> addGroupOID (this, oid);
}

...
```

```
/* Exporting a proxy
 * Algorithm: create a proxy-candidate
 *            create a server
 *            connect them with a trap reference
 */

        void
wrProvider::giveProxy
       (const importRequest* ir0,
        proxyDesc* result)
        raises (refused)
{
    // cast the import request into its known type
    const wrImportRequest* ir = (const wrImportRequest*) ir0;

    wrServer *server = 0;
    wrProxy *candidate = 0;
    // create a server and a proxy candidate
    begin
        // create server, put it in group, and get its OID
        server = new wrServer (ir->termName);
        giveMyOID (server, 1);  // to create a group
        OID serverOID;
        AS->getOID (server, &serverOID);

        // create candidate
        begin
            candidate = new wrProxy (serverOID, ir->termName);
        except
            when (noCode) {
            ...
            }
        end

        // prepare the candidate for exportation:
        // make candidate a member of the group
        giveMyOID (candidate, 1);
        // allow proxy to invoke server
        candidate->setTrapRef (server);
        // candidate will migrate and become proxy
        candidate->giveSelf (result);

    // many things can go wrong: can receive exceptions
    // cantOpenTty, tooBig, notFound, noCode, etc.
    // in all cases there is not much we can do.
    except
        when (others)  {
```

```
            if (server) delete server;
            if (candidate) delete candidate;
            raise (refused);
        }
    end

}
```

Chapter 9

The Name Service

9.1 Introduction

The Name Service of SOS allows users to reference SOS Objects by symbolic names (character strings); it maps these symbolic names with SOS references. The Name Service is realized by a set of name servers; each of them handles a part of the name space.

9.2 The name space

The name space is a single tree. A symbolic name is composed of name components which are directory names separated by the / character. Names beginning with a / are absolute names and refer to the root directory. All others are relative names and are interpreted relatively to the *current directory*.

The name's *authority* refers to the directory's name which manages it.

9.3 Name Service proxy importation

To use symbolic names, you have to import a proxy from the Name Service :

```
dirs* NS;

...

NS = new dynamic("/nameServer") dirs(nullIR);
```

9.4 Registering an object

The **addName** function allows to register an object name with the Name Service. The named object can be specified by its reference or by its address. The name is registered within one server which manages the authority for that name; if there's more than one server managing the authority, only one of them is choosed to handle the new name.

The function takes three arguments :

- The first is the symbolic name of the object.
- The second is either the reference, or the object.
- Setting the optionnal third argument to 1 allows to overwrite the entry corresponding to the symbolic name, if this name already exists and doesn't refer to a directory.

This example presents three usages of the **addName** function.

```
// the object is specified by its reference
myClass* myObject1 = new myClass;
ref myref;
AS -> getReference(myObject1, &myref);
NS -> addName("/myObject1", myref);

// the object is specified by its address
myClass* myObject2 = new myClass;
NS -> addName("/myObject2", myObject2);

// overwrite myObject2 by myObject3 under symbolic name /myObject2
myClass* myObject3 = new myClass;
NS -> addName("/myObject2", myObject3, 1);
```

9.5 Creating a new directory

The **addDir** function creates a new directory in the naming tree. There is only one argument : the symbolic name of the directory.

If the named directory already exists, the entry is not affected and an exception is raised.

For example :

```
NS -> addDir("/myDirectory");
```

9.6 Removing a name

The **delName** function allows to remove the mapping between a name and a reference. If the name is managed by several servers, the mapping is removed on all the servers.

For example :

```
NS -> delName("/myObject");
```

9.7 Finding the reference of an object

When an object has been registered with the Name Service, its reference can be retrieved with the **lookup** function, which takes two arguments :

- The first is the symbolic name of the object.
- The second is a pointer to a **reference** structure which will be filled by the Name Service.

For example :

```
ref refObject;

NS -> lookup("/myObject", &refObject);
```

9.8 Reading the contents of a directory

The contents of a directory can be read using the **getDirEntries** function. That function fills a buffer with **NSentry** structures, each of them describing a directory entry.

An NSentry structure contains :

- **dep** : the length of the structure.
- **length** : the length of the entry's name.
- **type** : the type of the entry.
- **name** : the name of the entry.

The **getDirEntries** function takes three arguments; the first one is the name of the directory, the second one is a pointer to the buffer to be filled and the last is the expected number of entries. It returns the total number of entries of the directory.

This example presents the usage of **getDirEntries** to read four entries in the ''**/users**'' directory.

```
char* bf = new char[sizeof(NSentry) * 4];
int nb = 0;
```

```
nb = NS -> getDirEntries("/users", bf, 4);
NSentry * e = (NSentry *) bf;
for (int i = 0; i < 4; i++) {
     cout << e -> name;
     e = (NSentry*) ((char*) e + e->dep);
}
```

9.9 Changing the current directory

The current directory is the directory from which the relative names are interpreted. The **changeDir** function allows to change it. The Name Service proxy importation set the current directory to the root.

```
NS -> changeDir ("/myDirectory");
```

Chapter 10

Object storage

10.1 Vertical migration

Objects may have an active and a passive representation. Objects in the active state (i.e. in volatile contexts) are managed by the Acquaintance Service: this is the state of newly created objects. Active objects may be passivated, i.e. managed by the Object Storage Service.

The Storage Service is in charge of *vertical migration*, e.g movement between passive and active representations.

10.2 Permanent objects

To have a permanent representation, an object's class must derive from `permObject` (which itself derives from `sosObject`). Every active `permObject` is automatically associated with a single *storage object* which controls its internal structure.

To use a `permObject`, two separate operations are necessary: creation (or importation), and initialization. The first operation is performed by the constructor of `permObject`, the second by operation `initC()` (after creation) or `initA()` (after importation). A `permObject` is stored on disk only when its method `checkpoint` is called.

Here is an example of a simple permanent object class.

```
    dynamic class
myClass: public permObject {
    int  i;
  public:
    myClass(int ii) {i=ii; initC();}
    myClass(importRequest * ir): (ir) {initA();}
    . . .
```

```
};
```

In the following example, the function `f()` creates (using the first constructor) and, stores (using `checkpoint()`) a permanent object, and registers the stored version with the name service:

```
f(){
    // create a permanent object
    myClass * myObject = new myClass (1234);
    . . .

    // save the state of myObject
    myObject -> checkpoint();

    // get the reference of myObject as stored on disk
    ref myRef;
    myObject -> getPermRef (&myRef);

    // now register myRef with the Name Service
    NS->addName ( "/myDir/myName", myRef);
}
```

Later you will be able to activate the stored object, using its name.
For instance :

```
main(){
    // importation from storage
    importRequest myImportRequest;
    myClass* myObject = new dynamic ("/myDir/myName")
                        myClass (&myImportRequest);

    . . .
}
```

10.3 Composite objects

SOS permanent objects can be made of multiple segments. The storage of composite objects is handled by the system if you respect the following conditions:

1. Data for which you have a pointer in one of your segments defines an indirect segment (which has no imposed structure).

2. Pointers to indirect data must use the system type `permPtr`

Usage of such pointers, called pointers to permanent data (or simply `permPtr`'s), is controlled by the system, once the user has initialized them with the operation `setPtr`.

A **permPtr** is untyped; typed versions of **permPtr** are available by macro-generation.

For instance, the following creates class **myClass** with an indirect segment containing an object of type **myIndSeg**.

```
// define the structure of an indirect segment

class myIndSeg {
      int  j;
    public:
      myIndSeg (int ii) {j=ii;}
      myIndSeg (importRequest* ir) {}
      void print();
      . . .
  };

// define the "permPtr on myIndSeg" type
typedef myIndSeg * myIndSegp;
declare (gpermPtr,myIndSegp);
typedef gpermPtr(myIndSegp) myIndSegPtr;

// define permanent composite object type
    dynamic class
myClass: public permObject {

  public:
    int  i;
    myIndSegPtr indPtr; //permanent pointer to myIndSeg object

    // constructor to create
    myClass (int, int);
    // constructor to import from disk
    myClass(importRequest* ir) : (ir), indPtr(ir) {
        this -> initA ();
        }
    . . .
};

// to create an instance of myClass...
myClass::myClass (int ii, int jj)  {
    // ... initialize for creation, ...
    this -> initC ();
    // ... initialize direct segment, and ...
    i=ii;
    // ... create and initialize indirect segment, ...
    myIndSeg *tmp = new myIndSeg (jj);
    // ... and initialize permPtr to indirect segment.
    indPtr.setPtr (getHead (),
```

```
                tmp,
                sizeof (myIndSeg));
}
```

The following small program creates a permanent object with an indirect segment, of type **myObject**, stores it, and registers its name with the Name Service, before exiting:

```
main () {
    // create permanent object with indirect sgmt
    myClass * myObject = new myObject (123, 456);
    // store it
    myObject -> checkpoint ();
    // register its name
    ref myRef;
    myObject -> getPermRef (&myRef);
    NS -> addName ("/myObjectName", myRef);
    exit(0);
    }
```

For this composite object, the procedure **permObject::checkpoint()** will store, both the direct segment defined by **myClass**, and the indirect segment, defined by **myIndSeg**.

Activation is done as before, but only the direct segment is loaded at activation time. The indirect segment(s) will be loaded only if necessary, i.e. at the time they are accessed.

For instance the following program uses the previously stored object. At importation time, only the direct segment is loaded. When the indirect segment is first used, it is also loaded.

```
main () {
    // import proxy from name service
    dirs * NS = new dynamic ("/nameServer") dirs (nullIR);
    // load permanent object
    myClass * myObject =
        new dynamic ("/myObjectName") myObject (nullIR);
    // now its direct segment may be accessed
    cout << "Value:" << myObject->i << "\n";

    // first access to indirect segment loads it
    myIndSeg* s = myObject -> indPtr;
    s -> j += 2;

    // mark as modified
    indPtr -> mark();
    // further accesses are direct to memory
    cout << "Other value: " << s->j << "\n";

    // checkpoint now stores the modified value
    // of the indirect segment
    myObject -> checkpoint () ;
```

```
exit(0);
}
```

In this program, we have changed the value of the indirect segment after reading it from disk. The new value is not put onto disk until **checkpoint()** is called. Only those segments which were signalled as changed by calling the **mark()** procedure will be written to disk.

Chapter 11

Remote communication protocols

The Communication Service allows the programmer to choose among different protocols for communicating between proxies and servers on different machines. Currently, only the SOS Remote Procedure Call Protocol (SOS-RPC), and a reliable multicast protocol are available. The SOS-RPC is the default protocol; this means that when a proxy makes a cross-invocation to a remote server, by default, they communicate using the SOS-RPC. Other protocols will be added in the future. We have presented how to set up the network environment in Chapter 2. In this chapter, we outline how to deal with the SOS-RPC. We also present the use of the multicast in SOS.

11.1 SOS-RPC

A proxy and its server are bound through their trap references. When the proxy and its server are in the same site, the kernel manages cross-invocation between them. When the proxy is migrated to a remote site, it must be able to communicate with its server. In order to provide such a possibility, we establish a connection between the proxy and its server. This connection ensures the remote extension of the cross-invocation. This extension is named *SOS-RPC* protocol. How this connection is installed and used is transparent for the user.

11.2 How to deal with the multicast

The communication service provides a multicast support. In order to deal with the multicast, one must first get a proxy of the SOS-RPC manager. The

provider of the SOS-RPC proxy is named **/rpcMgr**. This proxy provides the **createFamily** method described below.

11.2.1 Family creation

A *family* is a set of related SOS Objects, which can communicate together by multicast. A family has a name and a family OID. The creator has to provide, the family name, the family OID, and the number of members per site. By default, we assume one member per site. Hereafter is the example of the creation of the Speak family.

```
#include "rpc.h"

const char* speakFamilyName = "/speakFamily";

speakProvider::speakProvider(){
 ...
 // gets a proxy of communication service
 RPC = new dynamic ("/rpcMgr") rpcMgrProxy( nullIR );
 ...
 // alocates a group value to the OID
 const unsigned long SPEAKFAMILY_OID = 14012;
 OID familyOID;
 familyOID.groupAllocate( SPEAKFAMILY_OID );
 ...
 // creates the family with the allocated OID, the name of the family
 // and the maximum number of members per site
 RPC->createFamily( familyOID, speakFamilyName, MAXMAXMEMBERS);
 ...
}
```

The **createFamily** operation is idempotent.

11.2.2 How to join a family

Only objects derived from the **sosFamily** class may join a family. The jonction takes place at the instanciation of the derived object. Hereafter is an example of the creation of a member of the speak family previously created.

```
class speakRpc : public sosFamily {
 ....
 public:
  speakRpc( char*, OID, const char* );
  void send(char*, int, OID);
  void stub(invokeMessage*, returnMessage*, segmentDesc**);
};
```

```
speakRpc::speakRpc(char* name, OID memberOID,
  char* familyName) : (memberOID, familyName)
{...}
```

The caller must specify the family name (here **familyName**), and the internal identifier of this member within its family (here **memberOID**). If the member OID is not provided, the concrete OID of this member is assumed to be its internal OID within its family.

11.2.3 How members communicate

The **sosFamily** object provides a procedure **fcrossInvoke** allowing the invocation of one particular member, or the set of all other members (except the caller). This procedure is the equivalent of the **crossInvoke** primitive. A call to the family may be either a selective call, or a non selective one, i.e. a *multicall.*

If this is a selective call, one has to specify the same arguments as for a **crossInvoke** primitive, plus the member OID of the specific member which is the callee. This call returns a pointer to a **returnMessage**. We give an example of a selective invocation by one speak client to a particular other one, here specified by the **thisOne** argument.

```
  void
speakRpc::send (char* msg, int len, OID thisOne)
{
    speakInvokeMessage args (spkSend, len);

    // put the messsage in a segment
    segmentDesc *buf[2];
    ...

    // send to a particular member
    fcrossInvoke(&args, buf, thisOne );
    ....
}
```

For a multicall, the first and second arguments have the same meaning as for selective call. The third argument must be equal to **nullOID**. The first reply is returned as the result of the multicall to the caller. Other replies are saved within the communication context.

11.2.4 How to get other replies

After the return of a multicast, the caller may get others replies from other partners by calling the **giveNextReply** procedure, provided by the **sosFamily** object.

Bibliography

[Abr87] Vadim Abrossimov. Exception handling in C++ programs. In *Un recueil de papiers sur le système d'exploitation réparti à objets SOS, Rapport Technique INRIA no. 84*, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), May 1987.

[Gau87] Philippe Gautron and Marc Shapiro. Two extensions to C++: a dynamic link editor, and inner data. In *Proc. C++ Workshop*, USENIX, Santa Fe NM (USA), November 1987.

[Jon85] M. B. Jones, R. F. Rashid, and M. R. Thomson. Matchmaker: and interface specification language for distributed processing. In *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*, pages 225–235, ACM, New Orleans LA (USA), January 1985.

[Sha86a] Marc Shapiro. SOS: a distributed object-oriented operating system. In *2nd ACM SIGOPS European Workshop, on "Making Distributed Systems Work"*, Amsterdam (the Netherlands), September 1986. (Position paper).

[Sha86b] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, IEEE, Cambridge, Mass. (USA), May 1986.

[Sha86c] Marc Shapiro and Sabine Habert. Un système d'exploitation orienté objets pour SOMIW. In *3èmes Journées d'Étude Langages Orientés Objet*, AFCET, Paris (France), January 1986.

[Sha87a] Marc Shapiro, Vadim Abrossimov, Philippe Gautron, Sabine Habert, and Mesaac Mounchili Makpangou. *Un recueil de papiers sur le système d'exploitation réparti à objets SOS.* Rapport Technique 84, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), May 1987.

[Sha87b] Marc Shapiro, Vadim Abrossimov, Philippe Gautron, Sabine Habert, and Mesaac Mounchili Makpangou. SOS : un système d'exploitation réparti basé sur les objets. *Techniques et Sciences Informatiques*, 6(2):166–169, 1987.

[Str85] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1985.

[Str87] Bjarne Stroustrup and Jonathan E. Shopiro. A set of C++ classes for co-routine style programming. In *Proceedings and additional Papers, C++ Workshop*, USENIX, Berkeley, CA (USA), November 1987.

Appendix A

Glossary of terms

Acquaintance:
An SOS object activated and initialized in some context is called an acquaintance of that context.

Acquaintance Descriptor (AD):
An Acquaintance Descriptor is the data structure local to a context, which the Acquaintance Service uses to store information about an acquaintance. An Acquaintance Descriptor contains essentially: the array of the object's OID's, the array of its `trapRefs`, its prerequisites, the list of dependent objects, as well as its address.

Acquaintance Service (AS):
AS (ACQUAINTANCESERVICE[3]) is the SOS distributed Object Manager. Maintains Acquaintance Descriptors, and connections between objects (`trapReferences`). Supervises migration. Manages dependencies.

Candidate:
An object created or selected by a proxy provider, to be exported as a proxy.

Code Object:
A code object is an SOS object which serves as a carrier for the compiled procedures of a migratable class (see §6.1, CODE[2]). Initializing a code object consists of performing a dynamic link.

Constructor:
A C++ instance is initialized by a constructor for its class (which in turn calls a constructor for its base class and for included objects). If a constructor's first argument is of type `importRequest*` (or derived), it can be called only in an importation. (See also *Initialization* and *Procedure Table.*)

Context:
A context is a virtual memory space, containing objects. Each context

is initialized with a proxy for the Acquaintance Service. Communication within the context normally occurs by invocation. Communication between contexts is possible only via a proxy.

Cross Invocation:
A cross invocation is an invocation between two contexts of the same machine. (It can be continued onto an other machine by using the Communication Service.) The primitive CROSSINVOKE[2] is called by a proxy only, to call (one of) its server(s), designated by its trap reference. The system transports the invocation message and calls the server's `stub` procedure (STUB[2]).

Dependent:
An object's dependents are those objects which should be signalled automatically when an "important" (to be defined) state change occurs. When this happens, a previously-registered dependency procedure of each dependent is called. (((Not implemented yet.)))

Dependency Procedure:
See *dependent.*

Dynamic class:
A dynamic class is a class declared with the keyword `dynamic`, so that its code can be dynamically linked (see §3.3, DYNAMIC[2]).

Dynamic instance:
A dynamic instance is an instance of a dynamic class, which furthermore is instanciated using the `new dynamic` construct (see §4.1.1). Such an instance is an imported instance.

Dynamic constructor:
A dynamic instance is imported and re-initialized by applying a dynamic constructor, i. e. it is instantiated by `new dynamic`, and whose first argument is a `importRequest*` (or derived).

Dynamic linking:
When a code object is first imported into a context, the extant code is linked with it (see §6.1, DYNAMIC[2]). The newly imported code may now call, or be called by, the extant code [Gau87].

Elementary Object:
An elementary object is a localized piece of data, with associated code, accessed by invocation. An instance of a C++ class is an elementary object. An SOS object is an elementary object which is known to the SOS system; we often use the word "object" as a shorthand for "SOS object".

Exception:
An invocation can either return normally or with an exception (EXCEPTION[2]). In this case execution does not continue in sequence; control

may be regained by the program only by explicit action (**begin** ... **except** ... **when** ... **end** in SOS C++).

Exportation:
A provider may export an object to some other context. (((Currently, the only way to do this is for the target context to request an importation.)))

Group:
A Group (GROUP[2]) is a grouping of co-operating elementary SOS objects across contexts. It is a virtual communication space. The members of a group have at least one OID in common. For the system, a group is a distributed object; the members are incarnations of the group in some context, and are all considered equivalent. Communication within a group is unrestricted (whereas it is restricted outside of a group). The normal way of creating a group is to export objects, allocating a common OID to all of them.

Initialization:
When instantiating or importating an object, an initial procedure is invoked. For C++ objects, the initial procedure is a constructor. For an object created or imported under the control of the C++ compiler (the normal case), the compiler invokes one of its constructors. A migration can also be performed directly (e.g., migrating a prerequisite). In this case it is initialized by calling the procedure at index `SOS_CTOR_INDEX` (if defined) in the object's procedure table.

Importation:
Objects may be imported on request to the Acquaintance Service (see §4.1.1, DYNAMIC[2]). The AS forwards this request to a provider, which creates or selects object according to the request. Then the AS transports the object into the requesting context. Finally the object is initialized (again). See also *Migration*.

Invocation:
Invoking an object is calling one of its procedures. The normal invocation mode is local invocation, i.e. a simple procedure call. Invocation of an instance of a dynamic class is indirect via its procedure table. See also *Cross Invocation*.

Migration:
Some SOS objects can be migrated, i.e. exported from some context and imported into an other (see §3.3). When an object is migrated, this may cause migration of other, prerequisite objects. Migratable objects are instances of dynamic classes.

Name Service (NS):
NS (NAMESERVICE[5]) is the SOS distributed service for mapping symbolic names into references. Any object can be declared with the NS.

Object Identifier (OID):
An OID is a 64-bit integer (OID[2]). An elementary SOS object carries an array of Object Identifiers. The first (index 0) is its own, unique ID or "concrete OID". The others identify its group(s).

Object Storage Service (OSS):
OSS is the distributed service for storing passive images of objects onto disk (STORAGESERVICE[4]). In particular, Code objects may be stored with the OSS.

Object:
An object is either an elementary object or a group.

Prerequisite:
The prerequisites of an SOS object X are those objects Y, Z, ..., which must be present and initialized in the target context, before X is migrated and initialized (see §7.4). For instance, any object has its code as a prerequisite. (((Currently the general case is not implemented.)))

Principal:
An obsolete term for both *Provider* and *Server*.

Provider:
An object which exports other objects (i.e. proxies). Often, but not necessarily, the provider also acts as a server.

Procedure Table:
A per-object table of pointers to the `virtual` procedures, or (for a `dynamic` class) to all the procedures. Invocations occur indirectly via the procedure table.

Proxy:
A proxy is a local representative of a resource (PROXY[2]). The proxy may be an interface to one or more remote servers. A proxy is usually acquired dynamically, at the time of need, by an importation request directed to a provider. A proxy communicates with other members of the group by cross-invocation or any other means (e.g. shared memory).

Reference:
A reference or REF[2] is a fully-qualified, location-independent, fast object handle. It contains: an OID, a location hint, and an `opaqueField` which is used at the application's discretion. (((Subject to revision.))) When a reference is presented to the system, it searches for some object carrying the specified OID, using the hint to speed up the search.

Resource:
A service, implemented by a group, which is accessible to clients via a proxy interface. A resource is implemented by co-operation of providers (which create proxies in response to clients' requests), servers (which fulfill the service), and proxies (each client's local interface to the resource, and

local server). Any SOS object, suitably compiled and initialized, may fulfill any or all of these three rôles.

SOS object:
A C++ object declared to the system (see §3.1.1, SOSOBJECT[2])), by deriving its class from `sosObject`. See *elementary object.*

Server:
An object which responds to a proxy's requests.

Stub:
See *Cross Invocation.* The word "stub" is also used to name the communication code of both the caller and callee of a cross-invocation. See also *Stub Compiler.*

Stub Compiler:
A program which automatically generates the communication code for a cross-invocation (transforming data into an external representation and back, and initiating the communication) from a description of the invocation interface. (((Not implemented yet.)))

Trap Reference:
A `trapRef` of an object is a reference. It is a capability to some other object in its group, which it can cross-invoke.

Appendix B

Changes

B.1 Changes from V1 to V2

The syntax for indicating the provider has changed. In V1, it was part of the instance declaration:

```
dynamic (fileServer) file * f;
f = new file ();
```

In V2, it is part of the instantiation:

```
file * f;
importRequest ir = ...;
f = new dynamic (fileServer) file (&ir);
```

Existing services have been expanded, and new services added. SOS now runs distributed applications, using the Communication Service and a distributed version of the Acquaintance Service and of the Name Service.

Programs in V1 had to contain a

```
# include <sos/sos.h>
```

This is now reserved for operating-system programs. Application programs in V2 should do a

```
# include <sos/context.h>
```

B.2 Changes from V2 to V3

The new kernel allows to stop an individual application. A cross invocation to a terminated context raises `crossInvokeException`.

A new primitive **newContext** allows an application to start an other application. When contexts' pathnames are relative, they are searched from the directories listed in the Unix **PATH** environment variable (this affects **predefContexts** and NEWCONTEXT[2]).

Arguments passed to **sos** are transmitted to the contexts listed in the **predefContexts** file.

The maximum number of contexts running under SOS is raised to 16.

The **makexport** step for compiling migratable code and creating code objects doesn't exist any more; its functionality is subsumed by **sosCC**. Therefore there are no more **.e** files; all the necessary information is in the **.o** files. Now the code for a proxy can be linked statically with the client.

Applications can now import an object from remote storage.

The maximum length of file names must be of 13 characters for compatility with the Metaviseur.

The restriction that all **sosObject** instances must be created by **new** is raised. However, a proxy-candidate must be instantiated by **new**.

If the environment variable **DLPATH** is not set, the dynamic linker searches directory **/sos/export**.

The list of SOS hosts is now read from file **/sos/etc/sosHosts**.

A trace option **-T integer** has been added to **sos** and all contexts, to allow tracing of cross-invocations and/or task scheduling.

B.3 Changes from V3 to V4

The Unix filenames for the Sos environnement (e. g. **/sos/etc/predefContexts**) can be overriden with a environnement variable like PREDEFCONTEXTS. Furthermore, the value of the default is a compile-time option of the kernel and predefined services.

An object can cross-invoke an other object located in the same context. This is useful when the called object has migrated in the context of the calling object.

There is a new kernel more reliable.

The distribution management of the name space has changed. You can run SOS in any order on different machines with each proxy of the Name Service having the same view of the name space.

It is now possible to compile an SOS application with **gcc**, the GNU C Compiler without flag "-O". It provides faster code and smaller executables.

In dynamic class declarations, the compiler accepts list of file names. Nevertheless, the current version only performs importation of the first referenced name (see MERGEXPORT[1]).

Traces of dynamic classes (mentioned in comments) in the C file generated have changed (see chapter 7).

Interface of the class **code** has changed. This does not concern call to constructors, just methods to find code information (like the class name or the class

dynamic table), (see CODE[2]).

Run-time interface verification has been introduced. A key is supplied by the compiler (see DKEYOF[2]) and passed as extra argument to **sosImport** or **sosFindCode**. A null key disables the check. This modification should be transparent to users, but all files with dynamic class declarations have to be recompiled.

Facilities for dynamic linking debug have been introduced in the library `libD.a` (see DL_INFO[8]). The procedure **Zdebug()** is always available but the standard error (**stderr**) becomes the default output (see ZDEBUG[8]).

Appendix C

Compiling and debugging

In this chapter, we explain how to compile and how to debug SOS programs. We first show some examples of **sosObject** declarations. Then we talk about the compiler for SOS programs (**sosCC**). Debugging involves two sections; the first one deals with the debug of an executable SOS program, and the second one is about traces, a useful debug option to the **sos** program.

C.1 Declarations of sosObject

To use SOS objects, a program must include the file **context.h**. It contains the declarations needed for the execution of an SOS context.

```
#include "context.h"
...
```

Here are a few examples of declarations from a hypothetical C++ source file:

```
    dynamic
struct A: public sosObject { ... };

struct B {
   ...
   A *a1;        // OK
   A a2;         // OK (can't be exported)
   ...
   };

A *a3;          // OK
A a4;           // WRONG: static
B b1;           // WRONG: b1.a2 static
B *b3;          // OK
```

```
void p (A& a5){...}     // OK
void q (A *a6){...}     // OK
void r (A a7) {...}     // OK (can't be exported)
A&   s (...)            // OK
A    t (...)  {...}     // OK (can't be exported)

main () {
  static A a8;           // WRONG: static
  static B b3;           // WRONG: b3.a2 static
         A a9;           // OK (can't be exported)
         A* a10 = new A; // OK
         A* a11 = new dynamic A (...); // OK
  ...
  }
```

One cannot create **static** SOS objects. The examples marked "WRONG" in the program above will not be considered errors by the compiler, but will produce unpredicatble results at run-time.

Furthermore, automatic (i.e. stack-allocated) objects, and objects embedded within other objects, can't be exported (see Chapter 8). They are marked "can't be exported" above.

The keyword **dynamic** is explained in §3.3.

C.2 Termination and destructors

An SOS context, by default, never terminates. To terminate a context, it must execute the **exit(int)** procedure, with an integer parameter. By Unix convention, a zero value indicates that the context terminated successfully, and non-zero indicates an error. All the currently-executing tasks are immediately terminated. The detructors for static objects are then run. (Unfortunately, the destructors for non-static objects, allocated either on the stack or in free store, are not executed.) Finally the Unix process exits.

A context may also be killed from the outside by sending it a signal.

Terminating the **sos** program by sending it a signal **SIGTERM** (signal 15) has the effect of also immediately terminating all of the currently-running contexts. In this case, no destructors are run *at all.*

C.3 Compiling

Compilation is done by a special version of the C++ compiler `sosCC` (see sosCC[1]).

For instance, to compile a proxy, a provider, a client or a server, do this:

```
% sosCC -c myFile.c
```

This produces a linkable file **myFile.o**.
An executable SOS program (provider, client or server) is produced by :

```
% sosCC -o myProgram myFile.o otherFile.o ...
```

sosCC always links the necessary libraries, i.e **acquaintance.a**, **libD.a**, **kernel.a**, **libc.a**, **libC.a** and **directory.a**. Other libraries may also be linked with your executable (see §C.3.1).

C.3.1 Compiling a client

A client must be linked with all libraries needed by imported proxies. A small auxiliary file per library is needed to force inclusion of all of that library's definitions. For instance, if the proxy may need the **termcap** library, do this:

```
% dllib -o dtermcap.o /lib/libtermcap.a
% sosCC -c client.c
% sosCC -o client client.o dtermcap.o -ltermcap
```

The first line prepares the auxiliary file. The second one compiles **client.c**. The third links it statically with **dtermcap.o**, which forces every symbol in the **termcap** library to be linked.

When a client imports a proxy, this causes the code for the proxy do be dynamically linked, unless it is already present. Therefore, you can make executions faster, by linking the proxy code statically:

```
% sosCC -o client client.o otherFile.o proxy.o ...
```

Statically-linked proxies are also easier to debug.

C.4 Debugging

Debugging an SOS program can be done with an Unix debugger; it supposes you have previously compiled files with the **-g** option.

```
% sosCC -g -c myFile.c
```

However, it is more difficult to debug an SOS program which imports proxies. That is because Unix debuggers don't know how to read the symbol table of the proxy when its code is dynamically linked. So, for debug and when possible, it is wise to link statically the code of the proxy with the client.

```
% sosCC -o client client.o proxy.o ...
```

Dynamic debug facilities are supplied in the library **libD.a** (see DL_INFO[8]).
The procedure **Zdebug()** is available to list the dynamic classes currently linked (see ZDEBUG[8]).

```
#include "dyninfo.h"

main() {
   Zdebug();
   ...
}
```

If SOS crashes and won't restart correctly, kill SOS, delete the files `/sos/sosDsk/*/*` (or `SOSDSK/*/*`) and restart.

C.5 Traces

Unix debuggers only allow to trace the execution of a task inside a context; they don't permit to follow connections or cross-invocations between contexts and scheduling of tasks inside a context. An option to SOS programs permits to keep traces about context's activities, cross-invocations and scheduling of tasks.

An integer is associated with each type of trace: 1, for context's activities (connections between contexts through Unix sockets); 2, for cross-invocations (send and receive); and 4, for scheduling of tasks (choice of the task to be executed).

To obtain traces for each pre-defined context (i.e. each entry in the **predefContexts** file), run the **sos** program with the **-T** option followed by an integer, which is the sum of the traces values desired. For instance, to get full information :

```
% sos -T 7 &
```

The **-T** option is also available for any SOS context; for instance, to obtain information about cross-invocation activities in the **wrPrvdr** context, run it this way:

```
% wrPrvdr -T 2 &
```

Traces are written in the file **/tmp/xxx** where **xxx** is the Unix name of the context, followed by the Unix process-id of the context. For instance, in the previous example, traces are written to the file **/tmp/wrPrvdr1675** if **1675** is the process-id of **wrPrvdr**.

Appendix D

The write example

This chapter presents the code for the SOS "Write" example. This is part of the SOS distribution in the directory **examples/wr**. We divide this code in three parts, the proxy, the client and the provider. The **Makefile** shows the structure of the "Write" application:

```
# $Header: Makefile,v 3.2 88/03/29 19:50:09 shapiro Locked $

#CFLAGS= -I$(INCL) -c
CFLAGS= -O -c -I$(INCL) -I/usr/include/4.2
# where to find SOS
# SOSDIR==/sos==~sos/v3
SOSDIR=../..
INCL= $(SOSDIR)/include
BIN = $(SOSDIR)/bin
LIB = $(SOSDIR)/lib
EXP = $(SOSDIR)/export
CC = $(BIN)/sosCC

SOBJS= wrServer.o wrProvider.o wrProxy.o wrMain.o wrImportReq.o
COBJS= wrClient.o wrImportReq.o

all:    wrProxy.o Write wrPrvdr

all.dl: wrProxy.o Write.dl wrPrvdr.dl

# only works under sos
makecode: wrProxy.o
        makecode wrProxy wrProxy.o
```

```
install: all all.dl
#       install Write Write.dl wrPrvdr wrPrvdr.dl $(BIN)
        cp Write Write.dl wrPrvdr wrPrvdr.dl $(BIN)
        cp wrProxy.o $(EXP)

# write with static link
Write:  $(COBJS) wrProxy.o $(EXP)/dirproxy.o
        $(CC) -o Write $(COBJS) wrProxy.o $(EXP)/dirproxy.o

# Write with dynamic link
Write.dl: $(COBJS)
        $(CC) -o Write.dl $(COBJS)

# provider with static link
wrPrvdr:        $(SOBJS) $(EXP)/dirproxy.o
        $(CC) -DSOSTRACE -o wrPrvdr ${SOBJS} $(EXP)/dirproxy.o

# provider with dynamic link
wrPrvdr.dl:     $(SOBJS)
        $(CC) -DSOSTRACE -o wrPrvdr.dl ${SOBJS}

wrClient.o:     wrProxy.c wrClient.c
        $(CC) $(CFLAGS) wrClient.c


wrServer.o:     wrServer.h wrServer.c
        $(CC) $(CFLAGS) wrServer.c

wrProvider.o:   wrServer.h wrProvider.c
        $(CC) $(CFLAGS) wrProvider.c

wrMain.o:       wrServer.h wrMain.c
        $(CC) $(CFLAGS) wrMain.c

wrProxy.o:      wrProxy.h wrProxy.c
        $(CC) $(CFLAGS) wrProxy.c

wrImportReq.o:  wrProxy.h wrImportReq.c
        $(CC) $(CFLAGS) wrImportReq.c

clean:
        rm -f ${all} *.o *.s __err core a.out *.X


.SUFFIXES: .c .X .o .s
.c.X:
        $(CC) -I$(INCL) -F $*.c > $*.X
```

```
check: wrClient.X wrMain.X wrProvider.X wrProxy.X wrServer.X
```

D.1 The "Write" proxy

The "Write" proxy cross-invokes its server in the "Write" provider's context to write a client input on its destination terminal or to close the "Write" session. It performs some actions locally, for example, to print its state.

D.1.1 Proxy definitions

The definition of the proxy is in header file `wrProxy.h`:

```
static char wrProxy_rcsid[]="$Header: wrProxy.h,v 3.1 88/03/29 10:27:31 shapiro Locked :

static const int maxTermName = 80;
static const char wrProviderName[] = "/services/Write";

class wrImportRequest: public importRequest {
  public:
    char termName[maxTermName];
    wrImportRequest (const char* tty)
        raises (ttyNameTooLong);
};

dynamic class wrProxy: public sosObject {
    friend class wrProvider;

    OID serverId;
    char termName [maxTermName];
    wrProxy (OID&, const char*) // local constructor (for provider only)
        raises (tooBig);
public:
    wrProxy (wrImportRequest*); // importation constructor
    ~wrProxy();                 // destructor
    const char* getTermName () {return &termName[0];}

    int Write (char*, int);     // write msg on terminal
    void Quit();                // close connection
    void print();               // check state
};
```

D.1.2 Message definitions

The proxy communicates with the server using the message definitions in `wrInvoke.h`:

```
static char wrInvoke_rcsid[]="$Header: wrInvoke.h,v 3.1 88/03/29 10:27:10 shapiro Locked $";

// operation codes between wrProxy and its server
// (field opCode)

enum {wrSend, wrQuit};

// return codes (field retCode)

enum {wrOk=0, wrBadCodeOp=-1, wrFailed};

// parameters passed with cross-invocations

struct wrInvokeMessage: public invokeMessage{
    int     size;
    wrInvokeMessage (int op, int sz) { opCode = op; size = sz;};
    wrInvokeMessage (int op) {opCode=op; size=0;};
};

struct wrReturnMessage: public returnMessage {
    int result;
};
```

D.1.3 The code for the proxy

Finally, here is the code for the proxy itself. Note the presence of a constructor to create the candidate, for the private use of the provider, and another constructor for importation:

```
/*                                  wrProxy.c                                  */
static char rcsid[]="$Header: wrProxy.c,v 3.2 88/03/29 19:50:32 shapiro Locked $";

#include <stream.h>
#include "context.h"

#include "wrProxy.h"
#include "wrInvoke.h"

extern class dirs* NS;
static const char wrProxyCodeName[] = "/export/wrProxy.code";

/* importation constructor for the wrProxy:
 * there is nothing to do.
 * Remember to call the importation constructor of OID
 * so that the field serverId doesn't get over-written.
 */
wrProxy::wrProxy (wrImportRequest* ir):
        serverId(ir) {};


/* The creation constructor, for creating a proxy-candidate.
 * This is reserved to the provider.
 */
wrProxy::wrProxy (OID& id, const char* tty)
                raises (tooBig)
                raises (noCode)
{
    serverId = id;
    if (strlen (tty) > maxTermName)
        raise (tooBig);
    strcpy (termName, tty);

    // set code reference too code object
    ref wrCodeRef;
    begin
        NS->lookup (wrProxyCodeName, &wrCodeRef);
    except
        when (notFound)
            raise (noCode);
    end
```

```
    this -> setCodeRef (wrCodeRef);
}

/* Destructor.  Do nothing.
 * (In particular, don't do Quit: a destructor should not
 * cross-invoke.  The server will be signalled
 * automatically by dependencies -- in future versions!!)
 */
wrProxy::~wrProxy () {};

/* The client asks to write a message on the terminal.
 * This cross-invokes the server.
 */
        int
wrProxy::Write (char* msg, int len)
                raises (error)
                raises (closed)
{
    // if I already did a Quit, don't invoke server
    if (serverId == 0)
        raise (closed);

    // create and initialize invocation message
    wrInvokeMessage args (wrSend, len);

    // put the message in a segment
    segmentDesc *buf[2];
    buf[0] = new segmentDesc (0, sgCopyFrom);
    buf[0] -> assign (msg, len);
    buf[1] = 0;

    // cross-invoke server, and get answer
    wrReturnMessage* res =
        (wrReturnMessage*) crossInvoke (&args, buf);
    // clean up
    delete buf[0]; buf[0]=0;

    // decode answer
    if(res->retCode != wrOk)
        raise (error);
    return res->result;
}

/* The client asks to close the session.
   This invokes the server.
*/
         void
```

```
wrProxy::Quit()
{
    serverId = 0;                // make sure won't cross-invoke again

    wrInvokeMessage args (wrQuit);
    wrReturnMessage* res =
        (wrReturnMessage*) crossInvoke (&args);
    if (res-> retCode != wrOk)
        abort ();
}


/* Print a wrProxy intance, for debugging purposes.
*/
        void
wrProxy::print ()
{
    cerr << termName << ": ";
    serverId.print();
    cerr << "\n";
}
```

D.2 The "Write" client

The "Write" client takes a terminal name as argument:

```
% Write /dev/ttya
```

It first imports a proxy for this terminal from the "Write" provider, then loops on waiting user input to give to the "Write" proxy.

D.2.1 The import request

The file **wrImportReq.c** contains code to create the import request:

```
# include "context.h"
# include "wrProxy.h"

wrImportRequest::wrImportRequest (const char* tty)
    raises (ttyNameTooLong)
{
    if (strlen(tty) >= maxTermName) {
        raise (ttyNameTooLong);
    }
    else
        strcpy (termName, tty);
}
```

D.2.2 The client

The code of the client itself is in `wrClient.c`:

```
static char rcsid[]="$Header: wrClient.c,v 3.4 88/03/29 19:50:13 shapiro Locked $";

/*
 * wrClient.c
 */

#include <stdio.h>
#include <stream.h>
#include <fcntl.h>

extern char* getlogin();

#include "context.h"
#include "wrProxy.h"

// maximum size of user message + login name + 8
const int maxUserMsg = 128;

class dirs * NS = NULL;

/* Start a Write client, to send messages to another terminal.
 * Import a write proxy, from the provider.
 * Then wait for user input, and give to write proxy.
 */
main (int argc, const char* argv[])
{
    // must first import name service proxy
    NS = new dynamic ("/nameServer") dirs (nullIR);

    // skip options on command line
    for (int i=1; argc > 1; argc--)
        if (argv[i][0] != '-')
            break;
    // choose terminal to write to
    const char* tty;
    if (argc != 2) {
        tty = ttyname(2);
        cerr << argv[0] << ": writing to" << tty << "\n";
    } else
        tty = argv[1];

    // create request message, and ask for importation
    wrImportRequest args (tty);
```

```
    wrProxy *myWrProxy =
        new dynamic (wrProviderName) wrProxy (&args);
    cerr << argv[0] << " is ready (" << rcsid << ")\n";
    // if importation fails, the program will abort
    // due to uncaught exception

    // prepare message zone
    char buffer [maxUserMsg];
    strcpy (buffer, "From ");
    strcat (buffer, getlogin());
    strcat (buffer, ": ");
    const int prepared = strlen (buffer);
    char *bufp = buffer + prepared;

    // read user input & call proxy
    while (cin.get (bufp, maxUserMsg-1-prepared)) // read
    {
        char term;
        cin.get (term);         // eat terminator
        /* cout << "[" << buffer << "]\n";          // double check */
        // strlen+1 to include the final '\0'
        myWrProxy -> Write (buffer, strlen (buffer)+1);
    }

    // exited from loop: must be the end
    if (cin.eof()) {
        cout << "The end.\n";
    }
    myWrProxy -> Quit();

    delete myWrProxy;
    // must do exit or else context does not terminate
    exit (0);
}
```

D.3 The "Write" provider

It provides proxies for client applications, and creates a local server for each proxy. We show first the provider code. Declarations are in `wrProvider.h`:

```
static char wrProvider_rcsid[]="$Header: wrProvider.h,v 3.2 88/03/29 17:27:58 shapiro L

    class
wrProvider : public sosObject {
  public:
        wrProvider(unsigned long seed);
       ~wrProvider();
    void giveProxy (const importRequest*, proxyDesc*)
                    raises (refused);
};
```

D.3.1 Starting the provider

This file **wrMain.c** starts the provider:

```
static char rcsid[]="$Header: wrMain.c,v 3.3 88/03/29 19:50:23 shapiro Locked $";
// wrMain.c  main entry for the write server/provider

#include <stream.h>
#include "context.h"
#include "wrProvider.h"
#include "wrProxy.h"

dirs* NS = null;

main()
{

    const unsigned long wr_seed  = 14010; // Arbitrary

    // importation of a Name Service proxy
    NS = new dynamic ("/nameServer") dirs (nullIR);
    NS -> changeDir ("/");
    begin
        NS -> addDir ("/services");
    except
        when (nameExists)
            ;
    end

    // create the provider object
    wrProvider theProvider (wr_seed);

    // register with the Name Service
    ref tmp;
    AS->getReference (&theProvider, &tmp);
    NS->addName (wrProviderName, tmp, 1);

    cerr << "The Write provider is ready. (" << rcsid << ")\n";

    // make sure main doesn't return, so the provider remains allocated
    thistask -> resultis (0);

}
```

D.3.2 The provider

Here is the code for the provider itself, in file `wrProvider.c`:

```
/*                              wrProvider.c                    */
static char rcsid[]="$Header: wrProvider.c,v 3.3 88/03/29 19:50:27 shapiro Locked $";

#include <stream.h>
#include "context.h"
#include "wrProxy.h"
#include "wrProvider.h"
#include "wrServer.h"

/* Create a provider object.
*/
wrProvider::wrProvider(unsigned long seed)
{
    // allocation of a group OID for the Write service
    OID oid;
    oid.groupAllocate (seed);
    AS -> addGroupOID (this, oid);
}

/* Destructor.  Do nothing.
*/
wrProvider::~wrProvider () {}

/* Exporting a proxy
 * Algorithm: create a proxy-candidate
 *            create a server
 *            connect them with a trap reference
 */

        void
wrProvider::giveProxy
        (const importRequest* ir0,
         proxyDesc* result)
        raises (refused)
{
    // cast the import request into its known type
    const wrImportRequest* ir = (const wrImportRequest*) ir0;

    wrServer *server = 0;
    wrProxy *candidate = 0;
    // create a server and a proxy candidate
    begin
        // create server, put it in group, and get its OID
```

```
        server = new wrServer (ir->termName);
        giveMyOID (server, 1);  // to create a group
        OID serverOID;
        AS->getOID (server, &serverOID);

        // create candidate
        begin
            candidate = new wrProxy (serverOID, ir->termName);
        except
            when (noCode) {
                cerr << "No code object for write proxy!"
                     << "(Did you do makecode?)\n";
                if (server) delete server;
                if (candidate) delete candidate;
                exit (1);
            }
        end

        // prepare the candidate for exportation:
        // make candidate a member of the group
        giveMyOID (candidate, 1);
        // allow proxy to invoke server
        candidate->setTrapRef (server);
        // candidate will migrate and become proxy
        candidate->giveSelf (result);

    // many things can go wrong: can receive exceptions
    // cantOpenTty, tooBig, notFound, noCode, etc.
    // in all cases there is not much we can do.
    except
        when (others)  {
            cerr << "Write provider received exception!\n";
            if (server) delete server;
            if (candidate) delete candidate;
            raise (refused);
        }
    end

}
```

D.3.3 Server declarations

Finally, here is the server code. First, declarations, in `wrServer.h`:

```
static char wrServer_rcsid[]="$Header: wrServer.h,v 1.1 88/03/29 10:05:27 shapiro Locke

class wrServer : public sosObject {
    filebuf ttybuf;
    ostream *tty;
  public:
    wrServer (char*);
    ~wrServer ();
    void stub (invokeMessage*, returnMessage*, segmentDesc**);
};
```

D.3.4 The server

The code of the server itself is in **wrServer.c**:

```
/*                          file wrServer.c                          */

static char rcsid[]="$Header: wrServer.c,v 3.2 88/03/29 19:50:35 shapiro Locked $";

#include <stdio.h>
#include <fcntl.h>
#include <stream.h>

#include "context.h"

#include "wrProxy.h"
#include "wrInvoke.h"
#include "wrProvider.h"
#include "wrServer.h"

/* Create a server object (in the same context as Provider)
 * Opens a connection to the specified terminal
 */
wrServer::wrServer (char* ttyName)
    raises (cantOpenTty)
{
    if (ttybuf.open (ttyName, output) == 0)
        raise (cantOpenTty);
    tty = new ostream (&ttybuf);
}

/* Delete a server: close tty if not already done
 */
wrServer::~wrServer () {
    if (tty)
        delete tty;
}

/* Receive a cross-invocation, either to write a message
 * on the terminal, or to close the terminal.
 */
    void
wrServer::stub
    (invokeMessage* request,
     returnMessage* reply,
     segmentDesc** segs)
{
    wrInvokeMessage * req = (wrInvokeMessage*) request;
```

```
    wrReturnMessage * res = (wrReturnMessage*) reply;

    switch (request->opCode) {

    case wrSend: {              /* Send message to terminal */
        // don't need to check if tty still open
        // because proxy will never invoke after closing

        // map received segment to a character string
        const int size = req->size;
        char *msg = new char [ size ];
        segmentDesc segdesc (0, sgCopyTo);
        segdesc.assign (msg, size);
        segs[0] -> copyTo (&segdesc);

        // output  immediately
        *tty << msg << "\n";
        tty->flush ();

        // clean up & return
        delete [size] msg;
        res->result = tty->bad();
        if (res->result)
            res->retCode = wrFailed;
        else
            res->retCode = wrOk;
        break;
    }

    case wrQuit:                /* close connection */
        *tty << "The end.\n";
        delete tty; tty = 0;
        res -> result = 1;
        res -> retCode = wrOk;
        break;

    default:                    /* unknown operation */
        reply -> retCode = wrBadCodeOp;
    }
}
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique